

**A Collection of Topics**

**Compiled by**

**Tarık Sadak**

**React.JS**

# Chapter 1 : Introduction

## How much JavaScript do you need yo know?

You need a good understanding of the JavaScript language before reading this book. Nor will this book teach you anything about HTML/DOM and CSS. A good understanding of how these languages work is essential to getting good at React.

If you do not know what the DOM is, or understand the concept of children and parent nodes, or how to define simple event handlers on DOM elements, you are probably not ready for React. React can be used without understanding the DOM API, but you will be a lot more productive in React if you know the basics. If you don’t know anything about the DOM API, cover the short introduction to DOM presented here.

For JavaScript, if you do not know how to write **for** loops and **if** statements, or if you do not understand the concept of scopes and closures, you should probably learn these concepts in pure JavaScript before you dive into React. This book will introduce you to some adavanced and new concepts about Javascript first.

You don't have to be an expert, but I think you need a good overview of:

* Variables
* Arrow functions
* Work with objects and arrays using Rest and
* Spread
* Object and array destructuring
* Template literals
* Classes
* Callbacks
* Promises
* Async/Await
* ES Modules

In order to test the code snippets presented here you can use the simple jscomplete.com/playground or Visual Studio Code. If you use the first, all necessary imports are already loaded and you have two Dom elements to render your code: mountNode and MountNode2. Like:

mountNode.innerHTML = 'Hello World!';

or

ReactDOM.render(<h2>Hello React!</h2>, mountNode);

In the case of the latter, yu will need to install some software to be able to use React with VS Code, which will also be covered here.

## What is React?

Developed at Facebook and released to the world in 2013, it drives some of the most widely used apps, powering Facebook and Instagram among countless other applications.

Its primary goal is to make it easy to reason about a user interface and its state at any point in time, by dividing the UI into a collection of components.

React is used to build single-page applications and allows us to create reusable UI components.

**How did React get its name?**

When the state of a React component (which is part of its input) changes, the user interface it represents (its output) changes as well. This change in the description of the UI has to be reflected in the device we are working with. In a browser, we need to update the HTML DOM tree. In a React application, we don’t do that manually. React will simply react to the state changes and automatically (and efficiently) update the DOM when needed – hence the name.

React in itself has a very small API, and you basically need to understand four concepts to get started:

* Components
* JSX
* State
* Props

React is defined as a JavaScript library for building and simplify user interfaces. Let’s talk about the two different parts of this definition:

## React as a JS Library

It is not exactly a "framework." It is not a complete solution, and you will often need to use more libraries with React to form any solution. React does not assume anything about the other parts in any solution.

**So what is a framework and what is a library?**

While the terms Library and Framework may sound similar, they both work differently. Many people use these two words interchangeably without knowing the profound meaning behind them.

Before we dig into the key differences between Library and Framework, let's look at the common purpose that they both serve.

Both Library and Framework are code written by some developer to solve a complicated problem efficiently. They both give you an excellent approach to write DRY (don't repeat yourself) code. Their purpose was to increase the reusability of the code so that you can use the same piece of code or functions again in your various project.

### What is Library?

A Library is a set of code that was previously written by a developer that you can call when you are building your project. In Library, you import or call specific methods that you need for your project. In simple words, a bunch of code packed together that can be used repeatedly is known as Library. Reusability is one of the main reasons to use libraries.

Let's understand this more clearly with the help of an example. Think of you as a carpenter who needs to build a table. Now, you can build a table without the help of tools, but it's time-consuming and a long process. Whereas, if you choose the correct tools, you'll be able to build a table more quickly and that too without any hardship. Think of these tools as a library. You can write your program without them. But it will be a long process, and chances are your program will get buggy; while if you use Library, it'll be much easier for you to work with the program.

For example, if you use the built-in JavaScript fetch() method to fetch the data from API and you feel that it's not the ideal solution. Then you can use [Axios](https://www.npmjs.com/package/axios) Library for the same purpose to make your work easier.

axios.post('/login', {

    firstName: 'Monica',

    lastName: 'robinson '

})

    .then((response) => {

        console.log(response);

    }, (error) => {

        console.log(error);

    });

Some common examples of Library are:

**React** React is a JavaScript library for building user interfaces.

**Redux** Redux is an open-source JavaScript library for managing application state.  
It's most commonly used with React

**Three.js** It's another super cool JavaScript library used to create and display 3d computer graphics.

**Lodash** Lodash is a JavaScript library that provides utility functions for common programming tasks. It's more of a productivity kit in node.js

**jQuery** jQuery is a JavaScript library that does the things like event handling and HTML document manipulation.

### What is Framework?

A framework is a supporting structure that gives shape to your code. In the Framework, you have to fill the structure accordingly with your code. There is a specific structure for a particular framework that you have to follow, and it's generally more restrictive than Library.

One thing to remember here is that frameworks sometimes get quite large, so they may also use the Library. But the Framework doesn't necessarily have to use Library.

Let's get back to our carpenter and table example for a better understanding of the Framework. Here, if you want to build a table, then you need a model or skeleton for how the table looks, like the table has four legs and a top slab. Now, this is the core structure of the table and you have to work accordingly to build the table.

Similar to this, Framework also provides the structure, and you have to write the code accordingly.

Let's take the example of Express and understand the restrictive nature of the Framework.

var express = require('express')

var app = express()

app.get('/', function (req, res) {

    res.send('welcome to dev.to!! ')

})

app.post('/', function (req, res) {

    res.send('POST request to the dev.to homepage')

})

Here express is designed in such a way that it is going to look only for specific methods (get/post) and specific parameters.

You can't name the methods whatever you want to, and you have to name the methods as per the documentation.

Some common examples of Framework are:

**Angular** Angular is a JavaScript framework for web and mobile development.

**Django** Django is a fully featured server-side web framework written in Python.

**Express** Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

**Rails** Rails is a web application development framework written in the Ruby programming language.

**Spring** Spring Framework is an open-source framework for building web applications with Java as a programming language.

**Key Difference between Library vs Framework**

The main key difference between the Library and Framework is something known as inversion of control. Let's understand this inversion of control more in detail.

When you import a library, you have to call the specific methods or functions of your choice so, and it's up to you when and where to call the Library. Here, you are in charge of flow. On the other hand, Framework itself makes a call to your code and provide you with some space to write down details. So, while using framework your framework is in charge of flow. In Library, your code is going to call the Library whereas, in Framework, your code is being called by Framework.

Conclusion: I know it's getting too confusing but stay with me. I'll end this with one last crucial point. Here is a simple thing to remember: Framework is often more restrictive and generally have a more set of rules. Whereas, Library is not bounded by many rules.

## React as a UI Builder

When you hear the statement, "React is declarative," that is exactly what it means. We describe UIs with React and tell it what we want (not how to do it). React will take care of the "how" and translate our declarative descriptions (which we write in the React language) to actual UIs in the browser. React shares this simple declarative power with HTML itself, but with React we get to be declarative for HTML UIs that represent dynamic data, not just static data.

I think React was a game changer because it created a common language between developers and browsers that allows developers to declaratively describe UIs and manage transactions on their state, instead of transactions on their DOM elements. It’s simply the language of user interface "outcomes." Instead of coming up with steps to describe transactions on interfaces, developers just describe the interfaces in terms of a "final" state (like a function). When transactions happen to that state, React takes care of updating the UIs in the DOM based on that (and it does it efficiently, as we’ll see next).

If someone asked you to give one reason why React is worth learning, this outcome-based UI language is the one. I call this language "the React language."

## Why should you learn React?

1. React is very popular.
2. A lot of tooling today is built using React at the core.
3. As a frontend engineer, React is probably going to come up in a job interview.

It promotes several good development practices, including code reusability and components-driven development. It is fast, it is lightweight and the way it makes you think about the data flow in your application perfectly suits a lot of common scenarios.

## How does React work?

* React creates a VIRTUAL DOM in memory.
* Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.
* React only changes what needs to be changed!
* React finds out what changes have been made, and changes **only** what needs to be changed.

You will learn the various aspects of how React does this in the rest of this tutorial.

## How to Install React?

To start with, I highly recommend one approach, and that's using the officially recommended tool called:

create-react-app

*create-react-app* is a command line application, aimed at getting you up to speed with React in no time.

First of all you need npx and npm to run and create React applications, and they both come with Node.js. So that means you have to install node.js first on your computer. In order to accomplish this just go to <https://nodejs.org> and find the download package suitable for your computer.

If you have already installed Nodes, you can check which version you have by typing:

node -v

To check the version of npm you have installed, type:

npm -v

on command prompt. If you have an older version, you better update it to the latest.

This is all you need to have for a simple start for React, along with VS Code.

**Note:** If you've previously installed *create-react-app* globally, it is recommended that you uninstall the package to ensure npx always uses the latest version of create-react-app. To uninstall, run this command: *npm uninstall -g create-react-app*.

All things done, When you run:

npx create-react-app <app-name>

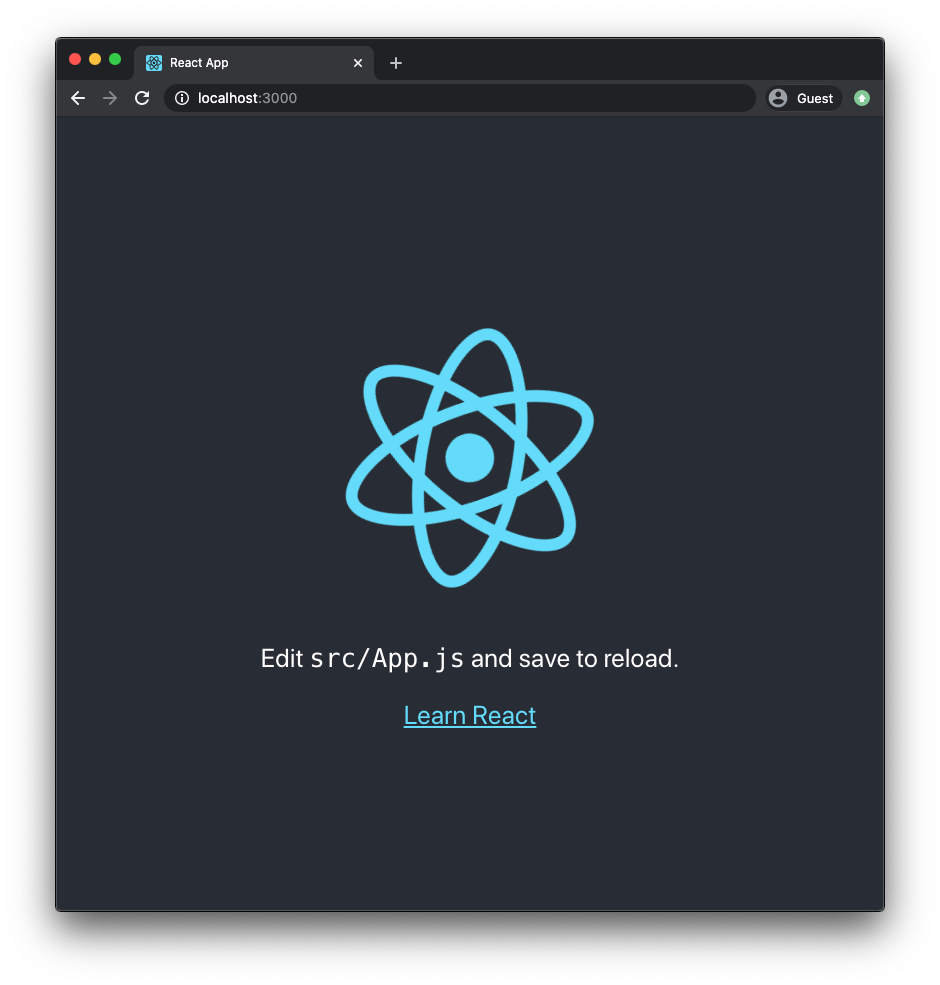
npx is going to download the most recent *create-react-app* release, run it, and then remove it from your system. This is great because you will never have an outdated version on your system, and every time you run it, you're getting the latest and greatest code available.

*create-react-app* includes built-in tools such as Webpack, Babel, and ESLint.

When you create a React application using the procedure described above, it will create an empty application which just produces the output on the next page. But in order to run the application you need to type the following from within the application directory (in the Terminal Window of VS Code or from command prompt):

npm start

After some notes about the compilation of the source code, what you will see is:



This is just like creating a new ASP.NET application. It’s running on a default port of :3000 and is just displaying this welcome screen with the React logo, and a comment directing you where to start building your application next (src/App.js); with a link to React’s website at the bottom.

## React.JS History

* Current version of React.JS is V17.0.2 (August 2021).
* Initial Release to the Public (V0.3.0) was in July 2013.
* React.JS was first used in 2011 for Facebook's Newsfeed feature.
* Facebook Software Engineer, Jordan Walke, created it.
* Current version of *create-react-app* is v4.0.3 (August 2021).
* ES6 stands for ECMAScript 6. ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015, and is also known as ECMAScript 2015.

# Chapter 2: A Pre-Course for React

Before starting with React.JS, you should have intermediate experience in:

* HTML
* CSS
* JavaScript

We will not teach these topics here. But you should also have some experience with the new JavaScript features introduced in ECMAScript 6 (ES6), you will learn about them in this chapter.

But before that, let’s go over a refresher of what DOM is. After that, we will present some important new aspects of ES6, which are commonly used in React.

## DOM

The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web.

### What is the DOM

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

Document Object Model (DOM) representation allows it to be manipulated. As an object-oriented representation of the web page, it can be modified with a scripting language such as JavaScript (as well as JQuery).

For example, the DOM specifies that the *querySelectorAll* method in this code snippet must return a list of all the <p> elements in the document:

const paragraphs = document.querySelectorAll("p");

// paragraphs[0] is the first <p> element

// paragraphs[1] is the second <p> element, etc.

alert(paragraphs[0].nodeName);

All of the properties, methods, and events available for manipulating and creating web pages are organized into objects. For example, the document object that represents the document itself, any table objects that implement the HTMLTableElement DOM interface for accessing HTML tables, and so forth, are all objects.

The DOM is built using multiple APIs that work together. The core DOM defines the entities describing any document and the objects within it. This is expanded upon as needed by other APIs that add new features and capabilities to the DOM. For example, the HTML DOM API adds support for representing HTML documents to the core DOM, and the SVG API adds support for representing SVG documents.

### DOM and Javascript

The document as a whole, the head, tables within the document, table headers, text within the table cells, and all other elements in a document are parts of the document object model for that document. They can all be accessed and manipulated using the DOM and a scripting language like JavaScript. Even if most web developers will only use the DOM through JavaScript, implementations of the DOM can be built for any other language.

### Accessing the DOM

For example, the following function creates a new <h1> element, adds text to that element, and then adds it to the tree for the document:

<html>

    <head>

        <script>

            {/\* run this function when the document is loaded \*/}

            window.onload = function() {

        /\*create a couple of elements in an empty HTML page\*/

        const heading = document.createElement("h1");

            const heading\_text = document.createTextNode("Big Head!");

            heading.appendChild(heading\_text);

            document.body.appendChild(heading);

      }

        </script>

    </head>

    <body>

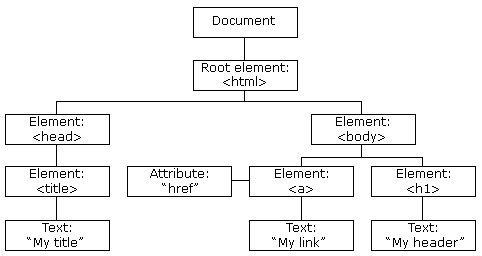
    </body>

</html>

### Fundemental Data Types in DOM

|  |  |
| --- | --- |
| Document | When a member returns an object of type document (e.g., the  ownerDocument property of an element returns the document to which it belongs), this object is the root document object itself. |
| Node | Every object located within a document is a node of some kind. In an  HTML document, an object can be an element node but also a text node or attribute node. |
| Element | The element type is based on node. It refers to an element or a node of type element returned by a member of the DOM API. Rather than saying, for example, that the document.createElement() method returns an  object reference to a node, we just say that this method returns the element that has just been created in the DOM. Element objects implement the DOM Element interface and also the more basic Node interface, both of which are included together in this reference. In an HTML document, elements are further enhanced by the HTML DOM API's HTMLElement interface as well as other interfaces describing capabilities of specific kinds of elements (for instance, HTMLTableElement for <table> elements). |
| NodeList | A nodeList is an array of elements, like the kind that is returned by the method document.querySelectorAll() . Items in a nodeList are accessed by index in either of two ways:   * list.item(1) * list[1]   These two are equivalent. In the first, item() is the single method on the nodeList object. The latter uses the typical array syntax to fetch the second item in the list. |
| Attr | When an attribute is returned by a member (e.g., by the createAttribute()  method), it is an object reference that exposes a special (albeit small) interface for attributes. Attributes are nodes in the DOM just like elements are, though you may rarely use them as such. |
| NamedNodeMap | A namedNodeMap is like an array, but the items are accessed by name or index, though this latter case is merely a convenience for enumeration, as they are in no particular order in the list. A namedNodeMap has an item() method for this purpose, and you can also add and remove items from a namedNodeMap. |

### Nodes vs Elements in the DOM

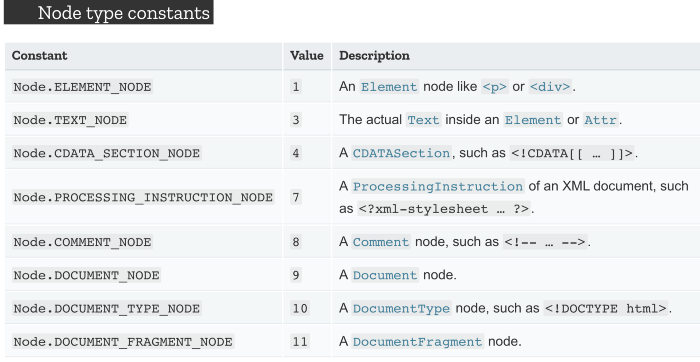


***What’s a node?***it is the name of any type of object in the DOM tree. It could be one of the built-in DOM elements such as the **document** itself, **document.head**or **document.body.**A node could be an HTML tag specified in the HTML such as **<input>**, **<div>,<h2>,** **<p>**or it could be a comment node, text node… In fact, a **node** is any DOM object and every node has a parent, every node is allowed to have one or more children or even zero children.

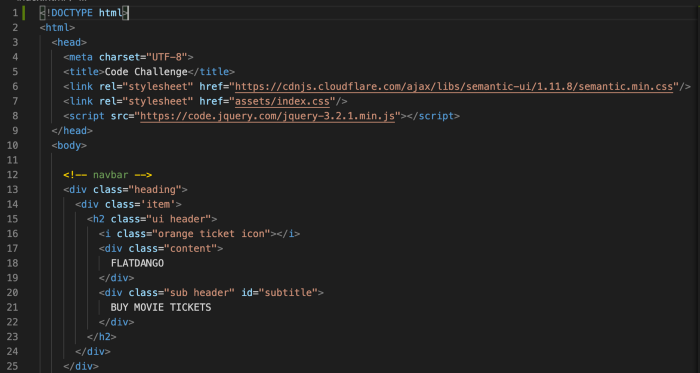
***What’s an element?***An element is a specific type of node, one that can be directly specified in the HTML with an HTML tag and can have properties like an **id** or a **class**. can have children, etc.

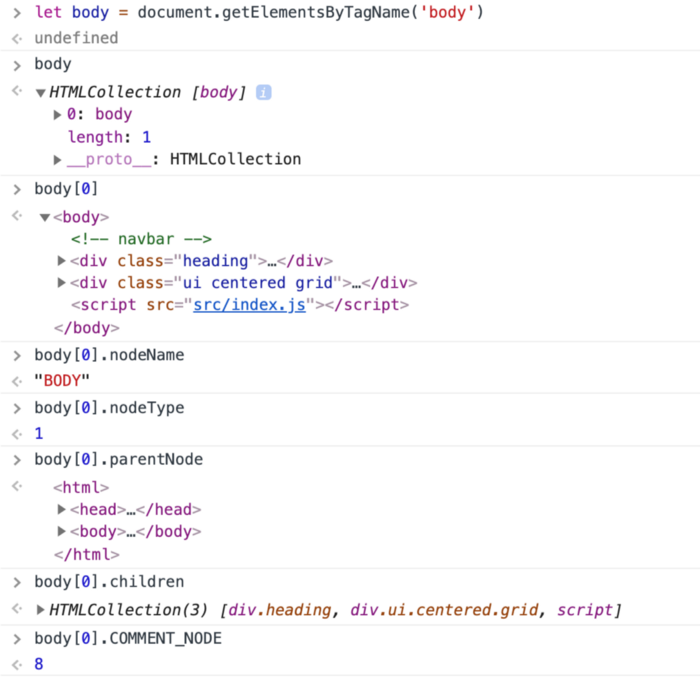
***Nodes vs Elements***: Nodes are all the different components that a webpage is made up of and elements are one type of node.

Let’s have a look on some types of node, their value and description:



The following html element has two children: head (line 3) and body (line 10) and these are two ‘nodes’. The head and body are element nodes. Inside the body (line 12) there is a comment (navbar), it’s one children of the body and it’s a comment node:





Since every node has properties, therefore let’s find out in the console the ones for element node ‘body’.

1. **.nodeName property**has a return value “Body”
2. **.nodeType**property returns a value of 1 which means that body is ELEMENT\_NODE.
3. **.parentNode**shows this element node “Body” has **parentNode**which is the html element.
4. **.children** property returns HTML collection,
5. **.COMMENT\_NODE** has a return value of 8, which states that we have comment node as of the children of the body.

### DOM Interfaces and Objects

When you get a reference to a table object, as in the following example, you routinely use all three of these interfaces interchangeably on the object, perhaps without knowing it.

const table = document.getElementById("table");

const tableAttrs = table.attributes; // Node/Element interface

for (let i = 0; i < tableAttrs.length; i++) {

    // HTMLTableElement interface: border attribute

    if (tableAttrs[i].nodeName.toLowerCase() == "border")

        table.border = "1";

}

// HTMLTableElement interface: summary attribute

table.summary = "note: increased border";

### Core Interfaces in the DOM

The document and window objects are the objects whose interfaces you generally use most often in DOM programming. In simple terms, the window object represents something like the browser, and the document object is the root of the document itself. Element inherits from the generic Node interface, and together these two interfaces provide many of the methods and properties you use on individual elements. These elements may also have specific interfaces for dealing with the kind of data those elements hold, as in the table object example in the previous section.

The following is a brief list of common APIs in web and XML page scripting using the DOM.

* document.querySelector(selector)
* document.querySelectorAll(name)
* document.createElement(name)
* parentNode.appendChild(node)
* element.innerHTML
* element.style.left
* element.setAttribute()
* element.getAttribute()
* element.addEventListener()
* window.content
* GlobalEventHandlers/onload
* window.scrollTo()

## Block scopes and the var/let/const keywords

### What is block scope & function scope?

A block scope is created with a pair of curly brackets. This happens every time you create an If statement, for statement, while statement, etc. The only exception is the curly brackets you use with functions. These create a function scope, not a block scope.

{

    // Block Scope

}

if (true) {

    // Block Scope

}

for (var i = 1; i <= 10; i++) {

    // Block Scope

}

function doSomething() {

    // Function Scope

}

### var

Function scopes are created for each function (like doSomething in Code Listing above). One difference they have from block scopes is obvious when using the **var** keyword. Variables defined with var inside a function scope are OK; they don’t leak out of that scope. If you try to access them outside of the scope, you can’t. (Show with an example in a function).

Before ES6 there were only one way of defining your variables: with the var keyword. If you did not define them, they would be assigned to the global object. Unless you were in strict mode, then you would get an error if your variables were undefined.

However, when you define variables with var in a block scope, you can totally access them outside that scope afterward, which is a bit problematic. For example, in a standard for-loop statement, if the loop variable is defined with var, you can access that variable after the loop is done. (Show with an example in a for loop).

So var has a *function* scope, not a *block* scope:

* If you use var outside of a function, it belongs to the global scope.
* If you use var inside of a function, it belongs to that function.
* If you use var inside of a block (but not a function block), i.e. a for loop, the variable is still available outside of that block.

### let

This is why the recommended way to declare variables in modern JavaScript is by using the **let** keyword instead of the var keyword. When defining variables with let, we won’t have this weird out-of-scope access problem. (Show with an example in a for loop).

However, you should use the let keyword only when the variable’s value needs to be changed. This should not be a common thing in your code. For most other cases you should use the **const** keyword instead, so let me tell you about it. (Show with an example in a for loop).

let v = {id : 42};

When you change the value of the variable v you are not really changing the content of the memory space that was initially associated with v. Instead, you’re creating a new memory space and changing the v label to be associated with that new space.

// Discard current memory unit (and its current label)

// Create new memory unit and label it as v

v = []; // No errors

### const

**const** is a variable that once it has been created, its value can never change.

const has a *block* scope.

The keyword const is a bit misleading. It does not define a constant value. It defines a constant reference to a value. Because of this you can NOT:

* Reassign a constant value
* Reassign a constant array
* Reassign a constant object

But you CAN:

* Change the elements of constant array
* Change the properties of constant object

When you use const to define a variable, you are instructing the computer to not only label a space in memory, but to also never change that label. The label will be forever associated with its same space in memory.

// Create a memory unit and label it as V

// This label cannot be discarded or reused

const V = { id: 42 };

// Later in the program

V = []; // TypeError: Assignment to constant variable.

Note that the constant part here is just the label. The value of what’s in the memory space can still change (if it’s mutable). For example, objects in JavaScript are mutable, so for the V in Code Listing above:

// You can do:

V.id = 37; // No errors

console.log(V.id); // 37

// But you still can't do:

V = { id: 37 }; // TypeError: Assignment to constant variable.

Strings and integers are immutable in JavaScript, so the only way to change a string or integer value in JavaScript is to discard the current memory space and re-label another one.

// Can't use const for this case:

let counter = 0;

counter = counter + 1; // Discard and re-label

Always use the **const** keyword to define variables. Only use the **let** keyword when you absolutely need it. Try to avoid the **var** keyword.

## Arrow Functions

Arrow functions are probably the most-used feature in modern JavaScript. Here’s what they look like:

const doSomething = () => {

    // Function Scope

};

This new "shorter" syntax to define functions is popular not only because it’s shorter, but also because it behaves more predictably with closures.

Arrow functions give access to their defining environment while regular functions give access to their calling environment. This access is possible through the special **this** keyword in a function’s scope:

* The value of the *this* keyword inside a regular function depends on how the function was called.
* The value of the *this* keyword inside an arrow function depends on where the function was defined.

Here is a code example to expand on the explanation:

this.whoIsThis = 'TOP' // Identify this scope

// Defining

const fancyObj = {

    whoIsThis: 'Fancy', // Identify this object

    regularF: function () {

        display.log('regularF', this.whoIsThis);

    },

    arrowF: () => {

        display.log('arrowF', this.whoIsThis);

    },

};

// Calling

display.log('TOP-LEVEL', this.whoIsThis); // It's "TOP" here

fancyObj.regularF(); // Output #1 (Fancy)

fancyObj.arrowF();   // Output #2 (Top)

fancyObj.regularF.call({ whoIsThis: 'Fake' }); // Output #3 (Fake)

fancyObj.arrowF.call({ whoIsThis: 'Fake' });   // Output #4 (Top)

This example has a regular function (regularF) and an arrow function (arrowF) defined in the same environment and called by the same caller. Here’s the explanation of the outputs in the last four lines:

* The regular function will always use its **this** to represent who called it. In the previous example, the caller of both functions was the fancyObj itself. That’s why Output #1 was Fancy.
* The arrow function will always print the this scope that was available at the time it was defined. That’s why Output #2 was TOP.
* The functions .call, .apply, and .bind can be used to change the calling environment. Their first argument becomes the new "caller." That’s why Output #3 was Fake.
* The arrow function does not care about the .call caller change. That’s why Output #4 was not changed and is still TOP, and not the new Fake caller.

One other cool thing about arrow functions is that if the function only has a single return line:

const square = (a) => {

    return a \* a;

};

You can make it even more concise by removing the curly brackets and the return keyword altogether.

const square = (a) => a \* a;

You can also remove the parentheses around the argument if the function receives a single argument:

const square = a => a \* a;

This much shorter syntax is usually popular for functions that get passed to array methods Like map, reduce, filter, and other functional programming methods:

console.log([1, 2, 3, 4].map(a => a \* a));

**Important:** Note that if you want to use the one-line version of the arrow function to make a function that returns an object, **you’ll have to enclose the object in parentheses**, because otherwise the curly brackets will actually be for the scope of the function.

// Wrong

const objMaker = () => { answer: 42 };

// Right

const objMaker = () => ({ answer: 42 });

## Closures

Closures can be a difficult concept to explain, so I will explain them through examples. Imagine a case where you want to write a function that can produce unique, incrementing numbers that can be used by other code in your web application. The only condition of this functionality is that if the last call to the function returned 10, the next call must return 11.

It is possible to write this functionality with a global variable:

> count = 0;

> function getNextCount() {

    return count++;

}

> getNextCount()

0

> getNextCount()

1

As has already been mentioned, however, global variables should be avoided because any other code can modify them.

These may look like contrived examples, but as web applications grow in size, global variables such as this become the source of difficult to find bugs. Closures provide an alternative.

Before looking at the solution, consider what happens to local variables inside a function when it finishes executing. The function that follows declares a local variable called myCount.

function counter() {

    var myCount = 0;

    return myCount++;

}

If you execute this and then attempt to access the variable, you will find it does not exist:

> counter()

0

> myCount;

ReferenceError: myCount is not defined

Now, consider this slight variation on the preceding function:

function getCounter() {

    var myCount = 0;

    return function () {

        return myCount++;

    }

}

Rather than returning a number, this function returns another function. The function that it returns has the following body:

function() {

    return myCount++;

}

You can now assign a variable to refer to this function:

counter = getCounter();

There is something strange about this function though: It is referring to the local variable myCount that was defined inside the function. Based on my previous explanation, this should have been destroyed when the call to getCounter finished. Therefore you might expect that if you invoke the function returned by getCounter, it will fail.

Not only does it not fail, it gives you exactly the behavior you want:

> counter();

0

> counter();

1

The anonymous function created inside getCounter is referred to as a closure. When it is created, it “closes” over all the variables in scope at the time, and obtains a reference to them. When the call to getCounter finished, therefore, JavaScript recognized that the anonymous function still might need to use the myCount variable and did not destroy it.

Although the anonymous function can continue to use the myCount variable, it is completely hidden from all other code. This means that it is not possible for any other code to interfere with the value of this variable:

> myCount = 10;

10

> counter()

2

The preceding code created a global variable called myCount, but this does not have any impact on your counter, which continues to use the local variable of the same name. In addition, if you were to create a second counter, it will have its own local myCount variable that will not impact your original counter. Instead, the new counter will also start counting from 0.

The beauty of this solution is that you have created private data. The function performing the counting is using a variable that only it has access to. This is an important technique in JavaScript because it does not support many of the mechanisms found in other languages for creating private data.

## Literal Notations

Modern JavaScript has a different way to define strings using the backtick character:

const html = `

    <div>

        ${Math.random()}

    </div>

`;

Strings defined with the backtick character are called template strings because they can be used as a template with dynamic values. They support string interpolation. You can inject any JavaScript expression within the ${} syntax.

With template strings, you can also have multiple lines in the string, something that was not possible with the regular-quoted strings. You can also "tag" template strings with a function and have JavaScript execute that function before returning the string, which is a handy way to attach logic to strings. This tagging feature is used in the popular styled-components library (for React).

## Expressions

In React, there is a syntax similar to the template literal syntax that you can use to dynamically insert a JavaScript expression into your React component’s code.

// Somewhere in a React component's return value

<div>

    {Math.random()}

</div>

This is NOT a JavaScript template literal. These curly brackets in React are how you can insert dynamic expressions in JSX. You don’t use a $ sign with them, although you can still use JavaScript template strings elsewhere in a React application (including anywhere within JSX curly brackets). This might be confusing, so here’s an example that uses both JSX curly brackets and JavaScript template literal curly brackets in the same line:

<div>

    {`Random value is: ${Math.random()}`}

</div>

The part in curly braces is the JavaScript template literal, which is an expression. We’re evaluating that expression within JSX curly brackets.

We will cover JSX later in detail.

## Destructuring (Arrays & Objects)

The destructuring syntax is simple, but it makes use of the same curly and square brackets you use with object or array literals, which makes it confusing sometimes. You need to inspect the context to know whether a set of curly brackets (**{}**) or square brackets (**[]**) are used as literal initializing or destructuring assignment.

const PI = Math.PI;

console.log({ PI });

const fn = ({ PI }) => { }

In Code Listing above, the first { **PI** } (in the second line) is an object literal that uses the **PI** constant defined in the first line. The second { **PI** } (in the last line) is a destructuring assignment that has nothing to do with the **PI** variable defined in the first line.

When destructuring arrays, the order that variables are declared is important. If we only want the car and suv we can simply leave out the truck but keep the comma:

const vehicles = ['mustang', 'f-150', 'expedition'];

const [car,, suv] = vehicles;

It can get a lot more confusing than that, but here is a simple general rule to identify what’s what:

When brackets appear on the left-hand side of an assignment or within the parentheses used to define a function, they are most likely used for destructuring. There are exceptions to this rule, but these exceptions are rare.

Here’s an example of destructuring:

// 1) Destructure array items

const [first, second, , fourth] = [10, 20, 30, 40];

// 2) Destructure object properties

const { PI, E, SQRT2 } = Math;

This is the same as:

// 1) assuming arr is [10, 20, 30, 40]

const first = arr[0];

const second = arr[1];

// third element skipped

const fourth = arr[3];

// 2)

const PI = Math.PI;

const E = Math.E;

const SQRT2 = Math.SQRT2;

This is useful when you need to use a few properties out of a bigger object. For example, here’s a line to destructure the **useState** and **useEffect** Hook functions out of the React API.

const { useState, useEffect } = React;

After this line, you can use these React API objects directly:

const [state, setState] = useState();

useEffect(() => {

    // do something

});

Note how the two items in the useState function’s return value (which is an array of exactly two items) were also destructured into two local variables. We’ll see examples of useState (and useEffect) in the upcoming chapters.

When designing a function to receive objects and arrays as arguments, you can also use destructuring to extract named items or properties out of them and into local variables in the function’s scope. Here’s an example:

const circle = {

    label: 'circleX',

    radius: 2,

};

const circleArea = ({ radius }, [precision = 2]) =>

    (Math.PI \* radius \* radius).toFixed(precision);

console.log(circleArea(circle, [5])); // 12.56637

The circleArea function is designed to receive an object in its first argument, and an array in its second argument. These arguments are not named, and not used directly in the function’s scope. Instead, their properties and items are destructured and used in the function’s scope. You can even give destructured elements default values (as it’s done for the precision item).

Another example:

const vehicleOne = {

    brand: 'Ford',

    model: 'Mustang',

    type: 'car',

    year: 2021,

    color: 'red'

}

myVehicle(vehicleOne);

function myVehicle({ type, color, brand, model }) {

    const message = 'My ' + type + ' is a ' + color + ' ' + brand + ' ' + model + '.';

}

## The rest/spread syntax

Destructuring gets more interesting (and useful) when combined with the rest syntax and the spread syntax, which are both done using the three dots (**...**). However, they do different things. The rest syntax is what you use with destructuring. The spread syntax is what you use in object/array literals.

Here’s an example:

const [first, ...restOfItems] = [10, 20, 30, 40];

The three dots here, because they are in a destructuring call, represent a rest syntax. We are asking JavaScript here to destructure only one item out of this array (the first one) and then create a new array under the name **restOfItems** to hold the rest of the items (after removing the first one).

This is powerful for splitting the array, and it’s even more powerful when working with objects to filter out certain properties from an object. For example, given this object:

const obj1 = {

  temp1: '001',

  temp2: '002',

  firstName: 'John',

  lastName: 'Doe',

  // many other properties

};

If you need to create a new object that has all the properties of **obj1** except for **temp1** and **temp2**, what would you do? You can simply destructure **temp1** and **temp2** (and ignore them), and then use the rest syntax to capture the remaining properties into a new object:

const { temp1, temp2, ...obj2 } = obj1;

How cool is that?

The spread syntax uses the same three dots to *shallow-copy* an array or an object into a new array or an object. This is commonly used to merge partial data structures into existing ones. It replaces the need to use the **Object.assign** method.

const array2 = [newItem0, ...array1, newItem1, newItem2];

const object2 = { ...object2, newP1: 1, newP2: 2, };

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

Example:

const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];

In React, the same three dots are used to spread an object of "props" for a component call. The JavaScript spread syntax was inspired by React (and others), but the usage of the three dots in React/JSX and in JavaScript is a little bit different. For example, given that a component **X** has access to an object like:

const engine = { href: "http://google.com", src: "google.png" };

That component can render another component **Cmpnt** and spread the properties of the **engine** object as props (attributes) for **Cmpnt**:

< **Cmpnt** {...engine} />

This is equivalent to doing:

< **Cmpnt** href={engine.href} src={engine.src} />

Note that the curly brackets in Code Listing above are the JSX curly brackets.

## Shorthand and dynamic properties

const mystery = 'answer';

const InverseOfPI = 1 / Math.PI;

const obj = {

    p1: 10,         // Plain old object property (don't abbreviate)

    f1() { },       // Define a shorthand function property

    InverseOfPI,    // Define a shorthand regular property

    f2: () => { },  // Define an arrow function property

    [mystery]: 42,  // Define a dynamic property

};

Did you notice **mystery**? It’s not an array or a destructuring thing—it’s how you define a dynamic property.

When you use the dynamic property syntax, JavaScript will first evaluate the expression inside **[]**, and whatever that expression evaluates to becomes the object’s new property.

In this example, the **obj** object will have a property **answer** with the value of **42**.

Another widely popular feature of object literals is available to you when you need to define an object with property names to hold values that exist in the current scope with the exact same names. You can use the shorthand property name syntax for that. That’s what we did for the **InverseOfPI** variable previously. That part of the object is equivalent to:

const obj = {

    InverseOfPI: InverseOfPI,

};

Objects are very popular in JavaScript. They are used to manage and communicate data, and using their modern literal features will often make your code a bit shorter and easier to read.

## Asynchronous Programming and Callbacks

Computers are asynchronous by design. Asynchronous means that a process (code or method) can run independently of and parallel to the main program flow.

Programs internally use *interrupts*, a signal that’s emitted to the processor to gain the attention of the system. Keep in mind that it’s normal for programs to be asynchronous, and halt their execution until they need attention, and the computer can execute other things in the meantime. When a program is waiting for a response from the network, it cannot halt the processor until the request finishes.

Normally, programming languages are synchronous, and some provide a way to manage asynchronicity, in the language or through libraries. C, Java, C#, PHP, Go, Ruby, Swift, Python, they are all synchronous by default. Some of them handle async by using threads, creating a new process.

**JavaScript**

JavaScript is synchronous by default and is single threaded. This means that code cannot create new threads and run in parallel.

Lines of code are executed in series, one after another. But JavaScript was born inside the browser, its main job, in the beginning, was to respond to user actions, like onClick, onMouseOver, onChange, onSubmit and so on. How could it do this with a synchronous programming model?

The answer was in its environment. The browser provides a way to do it by providing a set of APIs that can handle this kind of functionality.

More recently, Node.js introduced a non-blocking I/O environment to extend this concept to file access, network calls and so on.

**Callbacks**

You can’t know when a user is going to click a button, so what you do is, you define an event handler for the click event. This event handler accepts a function, which will be called when the event is triggered:

document.getElementById('button').addEventListener(

  'click',

  () => {

//code to execute when the button is clicked

  });

This is the so-called callback.

A callback is a simple function that’s passed as a value to another function, and will only be executed when the event happens. We can do this because JavaScript has first-class functions, which can be assigned to variables and passed around to other functions (called higher-order functions).

It’s common to wrap all your client code in a load event listener on the window object, which runs the callback function only when the page is ready:

window.addEventListener('load', () => {

  //window loaded

  //code to be executed when the windows is loaded

})

Callbacks are used everywhere, not just in DOM events. One common example is by using timers:

setTimeout(() => {

  console.log('Hello Timeout!');

}, 3000);

**Handling errors in callbacks**

How do you handle errors with callbacks? One very common strategy is to use what Node.js adopted: the first parameter in any callback function is the error object: error-first callbacks.

If there is no error, the object is null. If there is an error, it contains some description of the error and other information.

fs.readFile('/file.json', (err, data) => {

  if (err !== null) {

    //handle error

    console.log(err)

    return

  }

  //no errors, process data

  console.log(data)

})

**The problem with callbacks**

Callbacks are great for simple cases!

However every callback adds a level of nesting, and when you have lots of callbacks, the code starts to be complicated very quickly:

window.addEventListener('load', () => {

  document.getElementById('button').addEventListener('click', () => {

    setTimeout(() => {

      items.forEach(item => {

        //your code here

      })

    }, 2000)

  })

})

This is just a simple 4-levels code, but there can be much more levels of nesting and it’s not fun.

How do we solve this?

Starting with ES6, JavaScript introduced several features that help us with asynchronous code that do not involve using callbacks:

* Promises (ES2015)
* Async/Await (ES2017)

## Promises in Javascript

#### Introduction to promises

A promise is commonly defined as a proxy for a value that will eventually become available.

Promises are one way to deal with asynchronous code, without writing too many callbacks in your code.

Although they’ve been around for years, they were standardized and introduced in ES2015, and now they have been superseded in ES2017 by async functions.

Async functions use the promises API as their building block, so understanding them is fundamental even if in newer code you’ll likely use async functions instead of promises.

#### How promises work, in brief

Once a promise has been called, it will start in pending state. This means that the caller function continues the execution, while it waits for the promise to do its own processing, and give the caller function some feedback.

At this point, the caller function waits for it to either return the promise in a **resolved state**, or in a **rejected state**, but *the function continues its execution while the promise does it work*.

#### Creating a promise

The Promise API (Battery, Fetch, Service Workers API’s) exposes a Promise constructor, which you initialize using new Promise():

let done = true

const isItDoneYet = new Promise((resolve, reject) => {

  if (done) {

    const workDone = 'Work is finished'

    resolve(workDone)

  } else {

    const workNotDone = 'Still working on it'

    reject(workNotDone)

  }

})

As you can see the promise checks the done global variable, and if that’s true, we return a resolved promise, otherwise a rejected promise.

Using **resolve** and **reject** we can communicate back a value, in the above case we just return a string, but it could be an object as well.

#### Consuming a promise

In the last topic we introduced how a promise is created. Now let’s see how the promise can be used:

const isItDoneYet = new Promise()

//...

const checkIfItsDone = () => {

  isItDoneYet

    .then(ok => {

      console.log(ok)

    })

    .catch(err => {

      console.error(err)

    })

}

Running checkIfItsDone() will execute the isItDoneYet() promise and will wait for it to resolve, using the then callback, and if there is an error, it will handle it in the catch callback.

#### Chaining promises

A promise can be returned to another promise, creating a chain of promises.

A great example of chaining promises is given by the Fetch API, a layer on top of the XMLHttpRequest API, which we can use to get a resource and queue a chain of promises to execute when the resource is fetched.

The Fetch API is a promise-based mechanism, and calling fetch() is equivalent to defining our own promise using new Promise().

#### Example of chaining promises

const status = response => {

  if (response.status >= 200 && response.status < 300) {

    return Promise.resolve(response)

  }

  return Promise.reject(new Error(response.statusText))

}

const json = response => response.json()

fetch('/todos.json')

  .then(status)

  .then(json)

  .then(data => {

    console.log('Request succeeded with JSON response', data)

  })

  .catch(error => {

    console.log('Request failed', error)

  })

In this example, we call **fetch()** to get a list of TODO items from the **todos.json** file found in the domain root, and we create a chain of promises.

Running **fetch**() returns a response, which has many properties, and within those we reference:

* **status** , a numeric value representing the HTTP status code
* **statusText** , a status message, which is OK if the request succeeded

**response** also has a **json**() method, which returns a promise that will resolve with the content of the body processed and transformed into JSON.

So given those premises, this is what happens: the first promise in the chain is a function that we defined, called status() , that checks the response status and if it’s not a success response (between 200 and 299), it rejects the promise.

This operation will cause the promise chain to skip all the chained promises listed and will skip directly to the catch() statement at the bottom, logging the “Request failed” text along with the error message.

If that succeeds instead, it calls the json() function we defined. Since the previous promise, when successful, returned the response object, we get it as an input to the second promise.

In this case, we return the data JSON processed, so the third promise receives the JSON directly:

.then((data) => {

  console.log('Request succeeded with JSON response', data)

})

and we log it to the console.

#### Handling errors

In the above example, in the previous section, we had a catch that was appended to the chain of promises.

When anything in the chain of promises fails and raises an error or rejects the promise, the control goes to the nearest catch() statement down the chain.

new Promise((resolve, reject) => {

  throw new Error('Error')

}).catch(err => {

  console.error(err)

})

// or

new Promise((resolve, reject) => {

  reject('Error')

}).catch(err => {

  console.error(err)

})

#### Cascading errors

If inside the catch() you raise an error, you can append a second catch()to handle it, and so on.

new Promise((resolve, reject) => {

  throw new Error('Error')

})

  .catch(err => {

    throw new Error('Error')

  })

  .catch(err => {

    console.error(err)

  })

#### Orchestrating promises

If you need to synchronize different promises, **Promise.all()** helps you define a list of promises, and execute something when they are all resolved. Example:

const f1 = fetch('/something.json')

const f2 = fetch('/something2.json')

Promise.all([f1, f2])

  .then(res => {

    console.log('Array of results', res)

  })

  .catch(err => {

    console.error(err)

  })

The ES2015 destructuring assignment syntax allows you to also do

Promise.all([f1, f2]).then(([res1, res2]) => {

  console.log('Results', res1, res2)

})

You are not limited to using fetch of course, any promise is good to go.

**Promise.race()** runs as soon as one of the promises you pass to it resolves, and it runs the attached callback just once with the result of the first promise resolved.

Example:

const promiseOne = new Promise((resolve, reject) => {

  setTimeout(resolve, 500, 'one')

})

const promiseTwo = new Promise((resolve, reject) => {

  setTimeout(resolve, 100, 'two')

})

Promise.race([promiseOne, promiseTwo]).then(result => {

  console.log(result) // 'two'

})

Note: If you get the

Uncaught TypeError: undefined is not a promise

error in the console, make sure you use new Promise() instead of just Promise()

## Async/Await in Javascript

JavaScript evolved in a very short time from callbacks to promises (ES2015), and since ES2017 asynchronous JavaScript is even simpler with the async/await syntax.

Async functions are a combination of promises and generators, and basically, they are a higher level of abstraction over promises. Let me repeat: async/await is built on promises.

#### Why were async/await introduced?

They reduce the coding around promises, and the “don’t break the chain” limitation of chaining promises.

When Promises were introduced in ES2015, they were meant to solve a problem with asynchronous code, and they did, but over the 2 years that separated ES2015 and ES2017, it was clear that *promises could not be the final solution*.

Promises were introduced to solve the famous *callback hell* problem, but they introduced complexity on their own, and syntax complexity.

They were good primitives around which a better syntax could be exposed to developers, so when the time was right we got async functions.

They make the code look like it’s synchronous, but it’s asynchronous and nonblocking behind the scenes.

Debugging promises is hard because the debugger will not step over asynchronous code. Async/await makes this very easy because to the compiler it’s just like synchronous code.

#### How it works

An async function returns a promise, like in this example:

const doSomethingAsync = () => {

  return new Promise(resolve => {

    setTimeout(() => resolve('I did something'), 3000)

  })

}

When you want to call this function you prepend await, and the calling code will stop until the promise is resolved or rejected. One caveat: the client function must be defined as async. Here’s an example:

const doSomething = async () => {

  console.log(await doSomethingAsync())

}

#### A quick example

This is a simple example of async/await used to run a function asynchronously:

const doSomethingAsync = () => {

  return new Promise(resolve => {

    setTimeout(() => resolve('I did something'), 3000)

  })

}

const doSomething = async () => {

  console.log(await doSomethingAsync())

}

console.log('Before')

doSomething()

console.log('After')

The above code will print the following to the browser console:

Before

After

I did something //after 3s

#### Promise all the things

Prepending the async keyword to any function means that the function will return a promise.

Even if it’s not doing so explicitly, it will internally make it return a promise.

This is why this code is valid:

const aFunction = async () => {

  return 'test'

}

aFunction().then(alert) // This will alert 'test'

and it’s the same as:

const aFunction = async () => {

  return Promise.resolve('test')

}

aFunction().then(alert) // This will alert 'test'

#### The code is much simpler to read

As you can see in the example above, our code looks very simple. Compare it to code using plain promises, with chaining and callback functions.

And this is a very simple example, the major benefits will arise when the code is much more complex.

For example here’s how you would get a JSON resource, and parse it, using promises:

const getFirstUserData = () => {

  return fetch('/users.json') // get users list

    .then(response => response.json()) // parse JSON

    .then(users => users[0]) // pick first user

    .then(user => fetch(`/users/${user.name}`)) // get user data

    .then(userResponse => userResponse.json()) // parse J

  SON

}

getFirstUserData()

And here is the same functionality provided using await/async:

const getFirstUserData = async () => {

  const response = await fetch('/users.json') // get users list

  const users = await response.json() // parse JSON

  const user = users[0] // pick first user

  const userResponse = await fetch(`/users/${user.name}`)

  // get user data

  const userData = await userResponse.json() // parse JSO

  N

  return userData

}

getFirstUserData()

#### Multiple async functions in series

Async functions can be chained very easily, and the syntax is much more readable than with plain promises:

const promiseToDoSomething = () => {

  return new Promise(resolve => {

    setTimeout(() => resolve('I did something'), 10000)

  })

}

const watchOverSomeoneDoingSomething = async () => {

  const something = await promiseToDoSomething()

  return something + ' and I watched'

}

const watchOverSomeoneWatchingSomeoneDoingSomething = async () => {

  const something = await watchOverSomeoneDoingSomething(

  )

  return something + ' and I watched as well'

}

watchOverSomeoneWatchingSomeoneDoingSomething().then(res

  => {

  console.log(res)

})

This will print:

I did something and I watched and I watched as well

## Modules Import/Export

Modern JavaScript introduced the **import** and **export** statements to provide a solution for module dependency management," which is just a fancy term to describe JavaScript files that need each other.

You can export a function or variable from any file. There are two types of exports: Named and Default.

### Named Exports

You can create named exports two ways. In-line individually, or all at once at the bottom.

In-line individually:

export const name = "Jesse"

export const age = 40

All at once at the bottom:

const name = "Jesse"

const age = 40

export { name, age }

### Default Exports

You can only have one default export in a file.

const message = () => {

  const name = "Jesse";

  const age = 40;

  return name + ' is ' + age + 'years old.';

};

export default message;

You can import modules into a file in two ways, based on if they are named exports or default exports.

Named exports must be destructured using curly braces. Default exports do not.

**Import from Named Exports**

import { name, age } from "./xxx.js";

**Import from Default Exports**

import message from "./xxx.js";

#### More examples

A file **X.js** that needs to use a function from file **Y.js** can use the **import** statement to declare this dependency. The function in **Y.js** has to be *exported* first in order for any other files to import it. For that, you can use the **export** keyword:

export const functionY() { code }

Now, any file can import this named functionY export. If X.js is on the same directory as Y.js, you can do the following (or you can specify the relative path):

import { functionY } from './Y';

functionY();

The { **functionY** } syntax is not destructuring—it’s importing a named export.

You can also export without names using this syntax:

export default function () { //code }

When you import this default Y export, you can give it any name you want:

import function42 from './Y';

function42();

***Tip****: While default exports have their advantages, named exports play much better with intelligent IDEs that offer autocomplete, discoverability, and other features. It is usually better to use named exports, especially when you’re exporting many items in a module.*

## Conditional Expressions

Because you can only include expressions within the JSX curly brackets, you can’t write an **if** statement in them. You can, however, use a ternary expression:

<div>

  {condition ? valueX : valueY}

</div>

The short-circuit evaluation: It allows using a shorter syntax to put a value (or element) behind a condition by using the **&&** operator:

<div>

  {condition && <input />}

</div>

If **condition** is true, the second operand will be returned. If it’s false, React will ignore it. This means it will either render an input element or nothing at all. This JavaScript trick is known as the *short-circuit evaluation*.

## Timeouts & Intervals

Timeouts and intervals are part of a browser’s API. They’re not really part of the JavaScript language itself, but they’re used with JavaScript functions like **setTimeout** and **setInterval**.

Both of these functions receive a "callback" function and a "delay" value. setTimeoutwill invoke its callback function *once* after its delay value, while setIntervalwill *repeatedly* invoke its callback function with its delay value between each invocation.

This code will print the **Hello Timeout!** message after three seconds:

setTimeout(() => {

  console.log('Hello Timeout!');

}, 3000);

This code will print the Hello Interval! message every three seconds, forever:

setInterval(() => {

  console.log('Hello Interval!');

}, 3000);

A setInterval call will usually have an "exit" condition. Both setTimeout and setInterval return an id of the timer object they create, and that id value can be used to stop them. You can use a clearTimeout(id) call to stop a timeout object, and clearInterval(id) to stop an interval object.

This code will print the Hello Interval! message every three seconds, but only three times:

let count = 0;

const intervalId = setInterval(() => {

  count = count + 1

  console.log('Hello Interval!');

  if (count === 3) {

    clearInterval(intervalId);

  }

}, 3000);

## Classes

ES6 introduced classes, although React tends to abandon classes and use functions instead.

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

A simple class constructor:

class Car {

  constructor(name) {

    this.brand = name;

  }

}

Notice the case of the class name. We have begun the name, "Car", with an uppercase character. This is a standard naming convention for classes.

Now you can create objects using the Car class. Let’s create an object called "mycar" based on the Car class:

class Car {

  constructor(name) {

    this.brand = name;

  }

}

const mycar = new Car("Ford");

**Note:** The constructor function is called automatically when the object is initialized.

#### Method in Classes

You can add your own methods in a class. Let’s create a method named "present":

class Car {

  constructor(name) {

    this.brand = name;

  }

  present() {

    return 'I have a ' + this.brand;

  }

}

const mycar = new Car("Ford");

mycar.present();

As you can see in the example above, you call the method by referring to the object's method name followed by parentheses (parameters would go inside the parentheses).

#### Class Inheritance

To create a class inheritance, use the “extends” keyword.

A class created with a class inheritance inherits all the methods from another class. Let’s create a class named "Model" which will inherit the methods from the "Car" class:

class Car {

  constructor(name) {

    this.brand = name;

  }

  present() {

    return 'I have a ' + this.brand;

  }

}

class Model extends Car {

  constructor(name, mdl) {

    super(name);

    this.model = mdl;

  }

  show() {

    return this.present() + ', it is a ' + this.model

  }

}

const mycar = new Model("Ford", "Mustang");

mycar.show();

The super() method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

# Chapter 3: JSX

We have provided some code examples with JSX by far. Now let’s go into details.

We can't talk about React without first explaining JSX.

## What is JSX?

* JSX is an XML-based syntax for building DOM elements in JavaScript.
* JSX stands for **J**ava**S**cript **X**ML.
* JSX allows us to write HTML in React, that is to embed HTML directly in JS code.
* JSX makes it easier to write and add HTML in React.
* Under the hood, React will process the JSX and it will transform it into JavaScript that the browser will be able to interpret.

React gives us this interface for one reason: it's easier to build UIs using JSX.

## Coding JSX

JSX allows us to write HTML elements in JavaScript code and place them in the DOM without any createElement()  and/or appendChild() methods.

JSX converts HTML tags into React elements.

You are not required to use JSX, but JSX makes it easier to write React applications.

Here are two examples. The first uses JSX and the second does not:

#### Example 1: With JSX:

const myelement = <h1>I use JSX!</h1>;

ReactDOM.render(myelement, document.getElementById('root'));

Notice that the value is not a string. There are no quote marks around it. In fact, this is not valid JavaScript at all. If we try to run this as a JavaScript file, the browser's engine will crash. The browser does not understand JSX and neither does the React API.

Let us explain. First, let's discuss Modern JavaScript Bundling:

Since we're making applications in Front-End JavaScript now, our codebase is much bigger than it used to be. There are many files involved and thus a build step is needed to bundle them together. Without this step, an application might have 100s or 1000s of files which would need to be downloaded individually. One of the main roles of a bundler (like Webpack or Rollup) is to safely concatenate the files together to produce one file (or several files that are split up for application reasons). We get to author in many files that are organized for us to understand, but the end-result is built files that we put on production servers.

During the build process, plugins can be used to augment the code while it's being bundled. The most common plugin paired with React is Babel -- a JavaScript ***compiler*** which allows us to write JavaScript one way and Babel will rewrite it to be another way – which is called a ***transpiler***.

The reason we want Babel to rewrite our JavaScript is so that we can do things like use future features of JavaScript that browsers don't understand yet. Babel will rewrite our code to be compatible with what browsers currently understand. This makes it feel like we're writing future code and we don't have to wait for it to be ready in every browser.

So with Babel we write in JSX which will be re-written to be valid JavaScript:

// You get to write this JSX

const myDiv = <div>Hello World</div>

// And Babel will rewrite it to be this:

const myDiv = React.createElement('div', null, 'Hello World')

React doesn't deal with JSX directly. The React API expects to see React.createElement and not JSX at all. The idea is to have Babel transpile the JSX we write before React gets it.

We could write all our React code with React.createElement instead of using JSX if we wanted to. For this reason, JSX is said to be "Syntax Sugar", a term that means we can write in a nicer syntax instead of an alternative which isn't so nice.

*Babel's role is to intercept the files in the build process and to compile them accordingly. Their website even lets you*[*play around with example code to see how it would compile*](https://babeljs.io/repl#?babili=false&browsers=&build=&builtIns=false&spec=false&loose=false&code_lz=DwEwlgbgfAEgpgGwQewAQHVkCcEmAenGiA&debug=false&forceAllTransforms=false&shippedProposals=false&circleciRepo=&evaluate=true&fileSize=false&timeTravel=false&sourceType=module&lineWrap=false&presets=es2015%2Creact%2Cstage-2&prettier=false&targets=&version=7.3.4).

#### Example 2: Without JSX:

const myelement = React.createElement('h1', {}, 'I do not use JSX!');

ReactDOM.render(myelement, document.getElementById('root'));

As you can see in the first example, JSX allows us to write HTML directly within the JavaScript code.

Be careful about capitalization when using JSX. If you try to execute the **Button** function in a regular browser console, it’ll complain about the first character in the JSX part:

function Button(props) {

  //returns a Dom element here. Eg:

  return <button type="submit">{props.label}</button>; //label must be provided

}

JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime. But JSX is not HTML. JSX is not understood by browsers. What browsers understand (given the React library is included) is the React.createElementAPI calls. The same **Button** example can be written without JSX as follows:

function Button(props) {

  return React.createElement(

    "button",

    { type: "submit" },

    props.label

  );

}

ReactDOM.render(

  React.createElement(Button, { label: "Save" }),

  mountNode

);

JSX is basically a compromise. Instead of writing React components using the React.createElement syntax, we use a syntax very similar to HTML, and then use a transpiler to translate it into React.createElement calls.

A compiler that translates one form of syntax into another is known as a transpiler. To translate JSX, we can use transpilers like Babel or TypeScript. When you use create-react-app, the generated app will internally use Babel to transpile your JSX.

*Tip: You can use* ***babeljs.io/repl/*** *to see what any JSX syntax gets converted to for React; in fact JSX can also be used on its own, It is not a React-only thing.*

An example of this code translation:

<ul>

  {todos.map(todo =>

    <li>{todo.body}</li>

  )}

</ul>

Which, before being used in the browser, gets translated to:

React.createElement(

  "ul",

  null,

  todos.map(todo => React.createElement("li", null, todo.body)),

);

### Expressions in JSX

With JSX you can write expressions inside curly braces { }.

The expression can be a React variable, or property, or any other valid JavaScript expression. So it means that JSX is HTML in Javascript, but again you can use Javascript in JSX within curly brackets. JSX will execute the expression and return the result:

Example: Execute the expression 5 + 5:

const myelement = <h1>React is {5 + 5} times better with JSX</h1>;

Only expressions can be included inside these curly brackets. For example, you cannot include a regular if statement, but a ternary if expression is okay. Anything that returns a value is okay. You can always put any code in a function, make it return something, and call that function within the curly brackets. However, keep the logic you put in these curly brackets to a minimum. There are better places for long or complex logic, which we will see in the next chapter.

JavaScript variables are also expressions, so when the component receives a list of props you can use these props inside curly brackets. That’s how we used {props.label} (and {label}) in the Button example.

JavaScript object literals are also expressions. Sometimes we use a JavaScript object inside curly brackets, which makes it look like double curly brackets: { { a: 42 } }. This is not a different syntax; it is just an object literal defined inside the regular JSX curly brackets.

For example, one use case for using an object literal in these curly brackets is to pass a CSS style object to the special **style** attribute in React:

const ErrorDisplay = ({ message }) => (

  <div style={{ color: 'red', backgroundColor: 'yellow' }}>

    {message}

  </div>

);

ReactDOM.render(

  <ErrorDisplay

    message="These aren't the droids you're looking for"

  />,

  mountNode

);

The style attribute is a special one in React. We use an object as its value, and that object defines the styles as if we are setting them through the JavaScript DOM API (camel-case property names and string values). React translates these style objects into inline CSS style attributes. This is generally not the best way to style a React component, but I find it extremely convenient to use when applying conditional styles to elements. For example, here is a component that will randomly output its text in either green or red about half the time:

class ConditionalStyle extends React.Component {

  render() {

    return (

      <div style={{ color: Math.random() < 0.5 ? 'green' : 'red' }}>

        How do you like this?

      </div>

    );

  }

}

ReactDOM.render(<ConditionalStyle />, mountNode,);

#### Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

const myelement = (

  <ul>

    <li>Apples</li>

    <li>Bananas</li>

    <li>Cherries</li>

  </ul>

);

#### One Top Level Element

The HTML code must be wrapped in ONE top level element.

So if you like to write two paragraphs, you must put them inside a parent element, like a div element.

Example: Wrap two paragraphs inside one DIV element:

const myelement = (

  <div>

    <p>I am a paragraph.</p>

    <p>I am a paragraph too.</p>

  </div>

);

JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.

Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.

A fragment looks like an empty HTML tag: <></>. (Note: Instead of an empty tag you can also use *<React.Fragment>* tag – or even simply *<Fragment>*, but the empty tag is widely preferred)

Example. Wrap two paragraphs inside a fragment:

const myelement = (

  <>

    <p>I am a paragraph.</p>

    <p>I am a paragraph too.</p>

  </>

);

**Elements Must be Closed**

JSX follows XML rules, and therefore HTML elements must be properly closed. JSX will throw an error if the HTML is not properly closed.

Example: Close empty elements with />:

const myelement = <input type="text" />;

#### Attribute class = className

The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

Use attribute className instead. JSX solved this by using className instead. When JSX is rendered, it translates className attributes into class attributes.

Example: Use attribute className instead of class in JSX:

const myelement = <h1 className="myclass">Hello World</h1>;

**Note** that in order to embed a JavaScript expression inside JSX, the JavaScript must be wrapped with curly braces, {}.

## Using JSX to Compose UI

In particular, in a React component you can import other React components, and you can embed them and display them.

A React component is usually created in its own file, because that's how we can easily reuse it (by importing it) in other components.

But a React component can also be created in the same file of another component, if you plan to only use it in that component. There's no "rule" here, you can do what feels best to you.

To keep things simple let's create a component in the same App.js file.

We're going to create a WelcomeMessage component:

function WelcomeMessage() {

  return <p>Welcome!</p>

}

We're going to add it to the App.js file. Now inside the App component JSX we can add <WelcomeMessage /> to show this component in the user interface. We say WelcomeMessage is a **child component** of App, and App is its parent component. We're adding the <WelcomeMessage /> component as if it was part of the HTML language.

That's the beauty of React components and JSX: we can compose an application interface and use it like we're writing HTML.

## The Differences between JSX and HTML

* One of the differences might be quite obvious if you looked at the App component JSX: there's a strange attribute called className. But there's a problem. We are writing this UI code in a JavaScript file, and class in the JavaScript programming language is a reserved word. This means we can't use this reserved word as we want. It serves a specific purpose (defining JavaScript classes) and the React creators had to choose a different name for it. That's how we ended up with *className* instead of *class* .

Also, apart for the diffenrece in the keyword class, there are more attribute names that are different in JSX and HTML.

* JSX is not forgiving. If you forget to close a tag, you will have a clear error message. React usually gives very good and informative error messages that point you in the right direction to fix the problem.
* Another big difference between JSX and HTML is that in JSX we can embed JavaScript.
* **Adding custom styling to an element uses different syntax.** Inline styles are a little different. Take a look at the following HTML and its JSX equivalent:

*HTML →*<input style='border: 1px solid black; border-radius: 4px'/>

*JSX* → <input style={{border: '1px solid black', borderRadius: '4px'}}/>

* **You need to return a single parent element in JSX**. One of the major differences between HTML and JSX is that in JSX, you must return a single parent element, or it won't compile.

A lot of devs use <div>...</div>, but a better one that many people use is “fragment”, <>...</> which makes the code more readable.

* **Write all HTML Attributes in camelCase in JSX**. You need to write all HTML attributes and event references in camelCase while writing JSX.

So, onclick becomes onClick, onmouseover becomes onMouseOver, and so on:

## Embedding Javascript in JSX

Suppose the App component has a variable called message:

function App() {

  const message = 'Hello!'

  //...

}

We can print this value in the JSX by adding {message} anywhere in the JSX. Inside the curly brackets { } we can add any JavaScript statement, but *just one* statement for every curly bracket block.

## JSX is not a template language

Some JavaScript libraries that deal with HTML provide a template language for it. You write your dynamic views with an "enhanced" HTML syntax that has loops and conditionals. These libraries will then use JavaScript to convert the templates into DOM operations. The DOM operations can then be used in the browser to generate the DOM tree described by the enhanced HTML.

React (and ReactDOM) eliminated that step. We do not send the browser a template at all in a React application. We send it a tree of objects described with the React API. React uses these objects to generate the DOM operations needed to display the desired HTML tree.

*Note: With an HTML template, the library parses your application as a string. A React application is parsed as a tree of objects.*

While JSX might look like a template language, it really isn’t. It’s just a JavaScript extension that allows us to represent React’s tree of objects with a syntax that looks like an HTML template. In other words, it’s a **JSX is JavaScript with a different syntax**.Browsers don’t have to deal with JSX at all, and React does not have to deal with it either— only the compiler deals with it. What we send to the browser is template-free and JSX-free code.

In a React application, there is no template language at all.

In JSX we can use curly braces to inject variables:

// What we write

const message = 'Hello World'

const myDiv = <div>{message}</div>

// And Babel creates:

const message = 'Hello World'

const myDiv = React.createElement('div', null, message)

With a template language, this would be a way to do string interpolation. But that's not what's going on in JSX. The use of curlies lets us open JSX to the power of plain JavaScript. Instead of thinking of it as interpolation, think of it as passing an argument because that's exactly what you're doing.

When we write JSX, we can pass content into the body of the tag (between the opening and closing tag) which becomes the third argument to React.createElement. In fact, lets prove it by nesting elements:

// We write this:

const message = 'Hello World'

const myDiv = <div><span>{message}</span></div>

// Babel transpiles to:

const message = 'Hello World'

const myDiv = React.createElement('div', null,

  React.createElement('span', null, message)

)

The div gets turned into one React.createElement where the body of the div gets passed into the third argument. That argument is a span so it becomes it's own React.createElement. If we choose to not write JSX and to write React.createElement directly, we're going to end up with a not of nesting!

Another important takeaway is the part where the curly braces allows us to write JavaScript. We can do simple variables like message from above, or we can do more complex expressions:

// Some function that returns an array

const getMessage = function () {

  return ['Hello World']

}

// When we write this:

const myDiv = <div><span>{getMessage()[0]}</span></div>

// Babel changes it to this:

const myDiv = React.createElement('div', null,

  React.createElement('span', null, getMessage()[0])

)

The fact that we can call a function and then immediately use it's return array like this getMessage()[0] is not React at all – it's actually just JavaScript.

# Chapter 4: State & Lifecycle

This chapter introduces the concept of state and lifecycle in a React component. You can find a [detailed component API reference here](https://reactjs.org/docs/react-component.html).

Consider a ticking clock example. We call ReactDOM.render() to change the rendered output:

function tick() {

    const element = (

        <div>

            <h1>Hello, world!</h1>

            <h2>It is {new Date().toLocaleTimeString()}.</h2>

        </div>

    );

    ReactDOM.render(

        element,

        document.getElementById('root')

    );

}

setInterval(tick, 1000);

We will learn how to make the Clock component truly reusable and encapsulated. It will set up its own timer and update itself every second.

We can start by encapsulating how the clock looks:

function Clock(props) {

    return (

        <div>

            <h1>Hello, world!</h1>

            <h2>It is {props.date.toLocaleTimeString()}.</h2>

        </div>

    );

}

function tick() {

    ReactDOM.render(

        <Clock date={new Date()} />,

        document.getElementById('root')

    );

}

setInterval(tick, 1000);

However, it misses a crucial requirement: the fact that the Clock sets up a timer and updates the UI every second should be an implementation detail of the Clock.

Ideally we want to write this once and have the Clock update itself:

ReactDOM.render(

    <Clock />,

    document.getElementById('root')

);

To implement this, we need to add “state” to the Clock component.

State is similar to props, but it is private and fully controlled by the component.

## Converting a Function to a Class

You can convert a function component like Clock to a class in five steps:

1. Create an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes), with the same name, that extends React.Component.
2. Add a single empty method to it called render().
3. Move the body of the function into the render() method.
4. Replace props with this.props in the render() body.
5. Delete the remaining empty function declaration.

class Clock extends React.Component {

    render() {

        return (

            <div>

                <h1>Hello, world!</h1>

                <h2>It is {this.props.date.toLocaleTimeString()}.</h2>

            </div>

        );

    }

}

Clock is now defined as a class rather than a function.

The render method will be called each time an update happens, but as long as we render <Clock /> into the same DOM node, only a single instance of the Clock class will be used. This lets us use additional features such as local state and lifecycle methods.

## Adding Local State to a Class

We will move the date from props to state in three steps:

1. Replace this.props.date with this.state.date in the render() method:

class Clock extends React.Component {

    render() {

        return (

            <div>

                <h1>Hello, world!</h1>

                <h2>It is {this.state.date.toLocaleTimeString()}.</h2>

            </div>

        );

    }

}

1. Add a [class constructor](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor) that assigns the initial this.state:

class Clock extends React.Component {

    constructor(props) {

        super(props);

        this.state = { date: new Date() };

    }

    render() {

        return (

            <div>

                <h1>Hello, world!</h1>

                <h2>It is {this.state.date.toLocaleTimeString()}.</h2>

            </div>

        );

    }

}

Note how we pass props to the base constructor. Class components should always call the base constructor with props.

1. Remove the date prop from the <Clock /> element:

ReactDOM.render(

    <Clock />,

    document.getElementById('root')

)

We will later add the timer code back to the component itself.

The result looks like this:

class Clock extends React.Component {

    constructor(props) {

        super(props);

        this.state = { date: new Date() };

    }

    render() {

        return (

            <div>

                <h1>Hello, world!</h1>

                <h2>It is {this.state.date.toLocaleTimeString()}.</h2>

            </div>

        );

    }

}

ReactDOM.render(

    <Clock />,

    document.getElementById('root')

);

Next, we’ll make the Clock set up its own timer and update itself every second.

## Adding Lifecycle Methods to a Class

In applications with many components, it’s very important to free up resources taken by the components when they are destroyed.

We want to [set up a timer](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) whenever the Clock is rendered to the DOM for the first time. This is called “mounting” in React.

We also want to [clear that timer](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/clearInterval) whenever the DOM produced by the Clock is removed. This is called “unmounting” in React.

We can declare special methods on the component class to run some code when a component mounts and unmounts:

class Clock extends React.Component {

    constructor(props) {

        super(props);

        this.state = { date: new Date() };

    }

    componentDidMount() {

    }

    componentWillUnmount() {

    }

    render() {

        return (

            <div>

                <h1>Hello, world!</h1>

                <h2>It is {this.state.date.toLocaleTimeString()}.</h2>

            </div>

        );

    }

}

These methods are called “lifecycle methods”.

The componentDidMount() method runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

componentDidMount() {

    this.timerID = setInterval(

        () => this.tick(),

        1000

    );

}

Note how we saved the timer ID right on this (this.timerID).

While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that doesn’t participate in the data flow (like a timer ID).

We will tear down the timer in the componentWillUnmount() lifecycle method:

componentWillUnmount() {

    clearInterval(this.timerID);

}

Finally, we will implement a method called tick() that the Clock component will run every second. It will use this.setState() to schedule updates to the component local state:

class Clock extends React.Component {

    constructor(props) {

        super(props);

        this.state = { date: new Date() };

    }

    componentDidMount() {

        this.timerID = setInterval(

            () => this.tick(),

            1000

        );

    }

    componentWillUnmount() {

        clearInterval(this.timerID);

    }

    tick() {

        this.setState({

            date: new Date()

        });

    }

    render() {

        return (

            <div>

                <h1>Hello, world!</h1>

                <h2>It is {this.state.date.toLocaleTimeString()}.</h2>

            </div>

        );

    }

}

ReactDOM.render(

    <Clock />,

    document.getElementById('root')

);

Now the clock ticks every second.

Let’s quickly recap what’s going on and the order in which the methods are called:

1. When <Clock /> is passed to ReactDOM.render(), React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes this.state with an object including the current time. We will later update this state.
2. React then calls the Clock component’s render() method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock’s render output.
3. When the Clock output is inserted in the DOM, React calls the componentDidMount() lifecycle method. Inside it, the Clock component asks the browser to set up a timer to call the component’s tick() method once a second.
4. Every second the browser calls the tick() method. Inside it, the Clock component schedules a UI update by calling setState() with an object containing the current time. Thanks to the setState() call, React knows the state has changed, and calls the render() method again to learn what should be on the screen. This time, this.state.date in the render() method will be different, and so the render output will include the updated time. React updates the DOM accordingly.
5. If the Clock component is ever removed from the DOM, React calls the componentWillUnmount() lifecycle method so the timer is stopped.

## Using State Correctly

There are three things you should know about setState().

#### Do Not Modify State Directly

For example, this will not re-render a component. Instead, use *setState()*:

// Wrong

this.state.comment = 'Hello';

// Correct

this.setState({ comment: 'Hello' });

#### State Updates May Be Asynchronous

React may batch multiple setState() calls into a single update for performance.

Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

// Wrong

this.setState({

    counter: this.state.counter + this.props.increment,

});

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

// Correct

this.setState((state, props) => ({

    counter: state.counter + props.increment

}));

We used an [arrow function](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions) above, but it also works with regular functions:

// Correct

this.setState(function (state, props) {

    return {

        counter: state.counter + props.increment

    };

});

#### State Updates are Merged

When you call setState(), React merges the object you provide into the current state.

For example, your state may contain several independent variables:

constructor(props) {

    super(props);

    this.state = {

        posts: [],

        comments: []

    };

}

Then you can update them independently with separate setState() calls:

componentDidMount() {

    fetchPosts().then(response => {

        this.setState({

            posts: response.posts

        });

    });

    fetchComments().then(response => {

        this.setState({

            comments: response.comments

        });

    });

}

The merging is shallow, so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments.

#### The Data Flows Down

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn’t care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.

A component may choose to pass its state down as props to its child components:

<FormattedDate date={this.state.date} />

The FormattedDate component would receive the date in its props and wouldn’t know whether it came from the Clock’s state, from the Clock’s props, or was typed by hand:

function FormattedDate(props) {

    return <h2>It is {props.date.toLocaleTimeString()}.</h2>;

}

This is commonly called a “top-down” or “unidirectional” data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components “below” them in the tree.

If you imagine a component tree as a waterfall of props, each component’s state is like an additional water source that joins it at an arbitrary point but also flows down.

To show that all components are truly isolated, we can create an App component that renders three <Clock>s:

function App() {

    return (

        <div>

            <Clock />

            <Clock />

            <Clock />

        </div>

    );

}

ReactDOM.render(

    <App />,

    document.getElementById('root')

);

Each Clock sets up its own timer and updates independently.

In React apps, whether a component is stateful or stateless is considered an implementation detail of the component that may change over time. You can use stateless components inside stateful components, and vice versa.

# Chapter 5: React Componenets

**React is all about components.**

Components are like functions that return HTML elements.

You can think of components as simple functions (in any programming language). We call functions with some input, and they give us some output. We can reuse functions as needed and compose bigger functions from smaller ones.

React components are exactly the same; their input is a set of "props", and their output is a description of a user interface.

A React component can also have a private state to hold data that may change over the lifecycle of the component. This private state is an implicit part of the input that drives the component’s output, and that’s actually what gives React its name!

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

So, a React component is a JavaScript function that returns a React element (usually with JSX). When JSX is used, the **<tag></tag>** syntax becomes a call to **React.createElement("tag")**. It’s critically important for you to keep this in mind while building React components. You are not writing HTML— you are using a JavaScript extension to return function calls that create React elements (which are essentially JavaScript objects).

A component name must start with capital letter. This is really a requirement since we will be dealing with a mix of HTML elements and React elements. JSX will consider all names that start with a lowercase letter as names of HTML elements. This is important because HTML elements are passed as strings to **React.createElement** calls, while React elements need to be passed as variables:

Components come in two types, Class components and Function components, in this tutorial we will concentrate on Function components.

In older React code bases, you may find Class components primarily used. It is now suggested to use Function components along with Hooks, which were added in React 16.8. Therefore we will not deal with classes in details in this book.

## Class Component

A class component must include the extends React.Component statement. This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a render() method, this method returns HTML.

Example: Create a Class component called Car:

class Car extends React.Component {

  render() {

    return <h2>Hi, I am a Car!</h2>;

  }

}

React supports creating components through the JavaScript class syntax as well. Here is the same **Button** component example written with the class syntax:

class Button extends React.Component {

  render() {

    return (

      <button>{this.props.label}</button>

    );

  }

}

// Use it (same syntax)

ReactDOM.render(<Button label = "Save" />, mountNode);

In this syntax, you define a class that extends **React.Component**, which is one of the main classes in the React top-level API. A class-based React component has to at least define an instance method named **render**. This **render** method returns the element that represents the output of an object instantiated from the component. Every time we use the **Button** class-based component (by rendering a **<Button … />**), React will instantiate an object from this class-based component and use that object’s representation to create a DOM element. It’ll also associate the DOM-rendered element with the instance it created from the class.

Note how we used **this.props.label** inside the rendered JSX. Every component gets a special instance property named **props** that holds all values passed to that component’s element when it was instantiated. Unlike function components, the **render** function in class based components does not receive any arguments.

## Function Component

Here is the same example as above, but created using a Function component instead.

A Function component also returns HTML, and behaves much the same way as a Class component; but Function components can be written using much less code, are easier to understand, and will be preferred in this book.

Example: Create a Function component called Car:

function Car() {

  return <h2>Hi, I am a Car!</h2>;

}

Now your React application has a component called Car, which returns an <h2> element.

To use this component in your application, use similar syntax as normal HTML: <Car />

Example: Display the Car component in the "root" element:

ReactDOM.render(<Car />, document.getElementById('root'));

In its simplest form, a React component is a JavaScript function:

function Button(props) {

  // Returns a DOM/React element here. For example:

  return <button type="submit">{props.label}</button>;

}

// To render a Button element in the browser

ReactDOM.render(<Button label="Save" />, mountNode);

## Functions vs. Classes

Components created with functions used to be limited in React. The only way to make a component "stateful" was to use the class syntax. This has changed with the release of React Hooks, beginning with React version 16.8, which was released in early 2019. The React Hooks release introduced a new API to make a function component stateful (and give it many other features).

With this new API, most of what is usually done with React can be done with functions. The class-based syntax is only needed for advanced and rare cases. In this book, we’ll use the new Hooks-based API instead of the old class-based one. I believe the new API will slowly replace the old one.

I’ve used both APIs in large applications, and I can tell you that the new API is far superior to the old one for many reasons, but here are the ones that I personally think are the most important:

* The new API has more constraints. Constraints are good because they basically force you to write less-buggy code.
* You don’t have to work with class "instances" and their implicit state. You work with functions that are refreshed on each render. The state is explicitly declared, and nothing is hidden. All of this basically means that you’ll encounter fewer surprises in your code.
* You can separate any "stateful" logic into self-contained composable and sharable units. This makes it easier to break complex components into smaller parts. It also makes testing components easier.
* You can consume any stateful logic in a declarative way, and without needing to use any hierarchical "nesting" in components trees.

While class-based components will continue to be part of React for the foreseeable future, as a newcomer to the ecosystem, it makes sense for you to start purely with just functions (and Hooks) and focus on learning the new API (unless you have to work with a codebase that already uses classes).

## Components in Components

We can refer to components inside other components:

Example: Use the Car component inside the Garage component:

function Car() {

  return <h2>I am a Car!</h2>;

}

function Garage() {

  return (

    <>

      <h1>Who lives in my Garage?</h1>

      <Car />

    </>

  );

}

ReactDOM.render(<Garage />, document.getElementById('root'));

## Components in Files

React is all about re-using code, and it is recommended to split your components into separate files.

To do that, create a new file with a .js file extension and put the code inside it:

Note that the filename must start with an uppercase character.

Example: This is the new file, we named it "Car.js":

function Car() {

  return <h2>Hi, I am a Car!</h2>;

}

export default Car;

To be able to use the Car component, you have to import the file in your application.

Example: Now we import the "Car.js" file in the application, and we can use the Car component as if it was created here:

import React from 'react';

import ReactDOM from 'react-dom';

import Car from './Car.js';

ReactDOM.render(<Car />, document.getElementById('root'));

## App.js

When you create an empty React app, application comes with a series of files that do various things, mostly related to configuration, but there's one file that stands out: App.js .

App.js is the **first React Component** you meet. It has many lines, but in order to analyze it we will simplify this component’s code like below:

import React from 'react'

import logo from './logo.svg'

import './App.css'

function App() {

  return /\* something \*/

}

export default App

The things we import in this case are a JavaScript library (the npm package), an SVG image, and a React CSS file. Something that at first sight looks quite strange:

It looks like HTML but it has some JavaScript embedded into it. It’s JSX and we have talked about it in a whole chapter.

A component can have its own state, which means it encapsulates some variables that other components can't access unless this component exposes this state to the rest of the application.

A component can also receive data from other components. In this case we talk about **props**.

## Benefits of Components

The term "component" is used by many frameworks and libraries. We can even write web components natively using HTML5 features like custom elements and HTML imports. Components, whether we are working with them natively or through a library like React, have many advantages.

First, components make your code more readable and easier to work with. Consider this UI:

<a href="http://facebook.com">

  <img src="facebook.png" />

</a>

What does this UI represent? If you speak HTML, you can parse it quickly here and say, “it’s a clickable image.” If we’re to convert this UI into a component, we can just name it ClickableImage.

<ClickableImage />

We will see how to accomplisg this in the following topic.

## Reusing Components

React components can also be reused in the same application, and across multiple applications. For example, here’s a possible implementation of the **ClickableImage** component:

const ClickableImage = ({ href, src }) => {

  return (

    <a href={href}>

      <img src={src} />

    </a>

  );

};

Having variables for both the **href** and the **src** props is what makes this component reusable. For example, to use this component we can render it with a set of props:

<ClickableImage href="http://google.com" src="google.png" />

And we can reuse it by using a different set of props:

<ClickableImage href="http://bing.com" src="bing.png" />

***Tip****: In functional programming, we have the concept of pure functions (which do not have any observable side effects). They are basically protected against any outside state; if we give them the same input, we’ll always get the same output. If a React component does not depend on (or modify) anything outside of its definition (for example, if it does not use a global variable), we can label that component pure as well. Pure components have a better chance at being reused without any problems.*

**A bigger example**

We can think of HTML elements as built-in components in the browser. We can also use our own custom components to compose bigger ones. For example, let’s write a component that displays a list of search engines.

const SearchEngines = () => {

  return (

    <div className="search-engines">

      <ClickableImage href="http://google.com" src="google.png" />

      <ClickableImage href="http://bing.com" src="bing.png" />

    </div>

  );

};

Note how we used the **ClickableImage** component to compose the **SearchEngines** component!

We can also make the **SearchEngines** component reusable as well, by extracting its data into a variable and designing it to work with that variable.

For example, we can introduce a **data** array in a format like:

const data = [

  { href: "http://google.com", src: "google.png" },

  { href: "http://bing.com", src: "bing.png" },

  { href: "http://yahoo.com", src: "yahoo.png" }

];

Then, to make **<SearchEngines data={data} />** work, we just map the **data** array from a list of objects into a list of **ClickableImage** components:

const SearchEngines = ({engines}) => {

  return (

    <List>

      {engines.map(engine => <ClickableImage {...engine} />)}

    </List>

  );

};

ReactDOM.render(

  <SearchEngines engines={data} />,

  document.getElementById("mountNode")

);

This **SearchEngines** component can now work with any list of search engines we give to it.

## Props

Up to now we have seen many examples of props.

Props are arguments or initial values passed into React components. Props are passed to components via HTML attributes. “props” stands for **properties**. Passing props to components is a great way to pass values around in your application.

A component either holds data (has state) or receives data through its props.

React Props are like function arguments in JavaScript *and* attributes in HTML, and you send them into the component as attributes. To send props into a component, use the same syntax as HTML attributes:

Example: Add a "brand" attribute to the Car element:

const myelement = <Car brand="Ford" />;

The Car component receives the argument as a props object:

Example: Use the brand attribute in the component:

function Car(props) {

  return <h2>I am a { props.brand }!</h2>;

}

It’s a common practice to use object destructuring to get the props by name:

function Car({ brand }) {

  return <h2>I am a { brand }!</h2>;

}

Let’s remember this function we used previously:

function Button(props) {

  return React.createElement(

    "button",

    { type: "submit" },

    props.label

  );

}

ReactDOM.render(

  React.createElement(Button, { label: "Save" }),

  mountNode

);

**Pass Data**

Props are also how you pass data from one component to another, as parameters.

Example: Send the "brand" property from the Garage component to the Car component, as an attribute:

function Car(props) {

  return <h2>I am a {props.brand}!</h2>;

}

function Garage() {

  return (

    <>

      <h1>Who lives in my garage?</h1>

      <Car brand="Ford" />

    </>

  );

}

ReactDOM.render(<Garage />, document.getElementById('root'));

If you have a variable to send, and not a string as in the example above, you just put the variable name inside curly brackets:

Example: Create a variable named carName and send it to the Car component:

function Car(props) {

  return <h2>I am a {props.brand}!</h2>;

}

function Garage() {

  const carName = "Ford";

  return (

    <>

      <h1>Who lives in my garage?</h1>

      <Car brand={carName} />

    </>

  );

}

ReactDOM.render(<Garage />, document.getElementById('root'));

Or if it was an object:

Example: Create an object named carInfo and send it to the Car component:

function Car(props) {

  return <h2>I am a {props.brand.model}!</h2>;

}

function Garage() {

  const carInfo = { name: "Ford", model: "Mustang" };

  return (

    <>

      <h1>Who lives in my garage?</h1>

      <Car brand={carInfo} />

    </>

  );

}

ReactDOM.render(<Garage />, document.getElementById('root'));

**Note:** React Props are read-only! You will get an error if you try to change their value.

**How Babel gets into the picture:**

JSX was created to resemble HTML, but with the power of reusable components. By returning JSX from a function, we create that re-usability:

function User(props) {

  return <div>User: {props.name} is {props.age}</div>

}

// When a function (with a capital letter) returns JSX,

// it is said to be a component. As such, it can be used

// like an element like this:

function App() {

  return (

    <div>

      {/\* These are two User elements \*/}

      <User name="Dave" age="34" />

      <User name="Julie" age="25" />

    </div>

  )

}

Babel transpiles the <User /> element like this:

// Notice the first argument is a reference to the function instead

// of a string like it would be with a DOM based element (like 'div')

React.createElement(User, { name: "Julie", age: "25" })

We can see how the props are converted to an object. This object is passed as the first argument to the User function.

In the above example, we're passing string props. If we want to pass a value other than a string, we do curly braces:

<User name="Julie" age={25} />

Babel now transpiles to:

React.createElement(User, { name: "Julie", age: 25 })

Here's how we pass in values of different types:

<OurComponent

  name="Brad"                   // String

  age={34}                      // Number

  retired={false}               // Boolean

  trainer={true}                // Boolean

  hobbies={[                    // An array

    'Computer Stuff',

    'Long walks on the beach'

  ]}

  location={{                   // An Object

    state: 'Arizona',

    country: 'US'

  }}

  onRemove={removeUser}         // A reference to a function

  onSave={() => {               // An inline arrow function

    console.log('Hello')

  }}

/>

Pay special attention to arrays and objects, notice that the first set of curlies is telling JSX that we want to do some JavaScript. Then we can do square brackets for arrays or another set of curlies for an object literal.

**Tags**

Which of these two is correct?

// This:

<OurComponent />

// Or this:

<OurComponent></OurComponent>

They're actually both correct. Both in the example do the exact same thing. In fact, you can even write self-closing DOM elements like this: <div /> which is valid JSX.

**Children props**

A special prop is called . children. That contains the value of anything that is passed between the opening and closing tags of the component, for example:

<WelcomeMessage> Here is some message </WelcomeMessage>

In this case, inside WelcomeMessage we could access the value “Here is some message” by using the children prop:

function WelcomeMessage({ children }) {

  return <p>{children}</p>

}

Another example. If you want, you can pass a special kind of prop called **children** into an element:

<User age={56}>Brad</User>

We don't see that word children as a prop "attribute". Instead it's the body between the opening and closing tags. Whatever we put there is received as **props.children**. Here's how that User component might look:

function User(props) {

  // props.children is "Brad"

  return <div>User: {props.children} is {props.age}</div>

}

Although it's not common to do, the children prop can be passed like this too:

<User age={56} children="Brad" />

However, the conventional purpose of using children is to pass in more JSX:

// This is easy to read:

<User age={56}>

  <span>Brad</span>

</User>

// This works the same, but not as easy to read:

<User age={56} children={<span>Brad</span>} />

**Map and Keys**

Let's say you want to make a list of users from an array:

function UserList() {

  const users = ['Michael', 'Ryan']

  return (

    <ul>

      <li>{users[0]}</li>

      <li>{users[1]}</li>

    </ul>

  )

}

The two li elements are going to become the children props to the ul element. React needs children to be one thing though, so internally React makes this an array. We could mimic that by doing this:

function UserList() {

  const users = ['Michael', 'Ryan']

  return (

    <ul>

      {[

        <li>{users[0]}</li>,

        <li>{users[1]}</li>

      ]}

    </ul>

  )

}

This is just to demonstrate that we *can* do this, not that we would. We probably want to write JSX that more closely resembles HTML. But just keep in mind that we *could* do this.

Right now the code isn't very dynamic. Instead of hardcoding the user's array let's pass it through props where the element is invoked. Then we can iterate over the values using .map:

function UserList(props) {

  return (

    <ul>

      {props.users.map(name => {

        return <li>{name}</li>

      })}

    </ul>

  )

}

<UserList users={['Michael', 'Ryan']} />

Iterating with .map is perfect because we need to convert one array to another array -- an array of strings to an array of JSX. It's also perfect because .map returns a value. Remember, we need to pass a value into the curlies of JSX. This is the reason why we can't use .forEach because it does not return a value. This is also why we can't use things like while, if, switch, for, or forEach in JSX.

Even though the above examples work, we're missing something called a key.

When we write JSX and React build the DOM for us, there's a reconciliation process that React uses to track which of our components go with which parts in the DOM. React knows how to take care of that for us except in cases where we provide JSX as an array (which is what we're doing when we do .map). Therefore, we have to supply a unique key so React can track things:

// .map supplies the index as the second argument of

// the callback, which we can use for a key:

function UserList(props) {

  return (

    <ul>

      {props.users.map((name, index) => {

        return <li key={index}>{name}</li>

      })}

    </ul>

  )

}

A few notes on keys:

* The key just needs to be unique for the array, not for the whole component or the whole application. If you had one component that made a list of users and a list of pets, each list could use the index from the array and that's fine.
* Keys can be numbers or strings, as long as they're unique.
* You may hear that using indexes of the array is a bad idea. That's only true if the list is going to change while the component is still mounted. Imagine if we used indexes and our array had 3 items then one was removed. If the first item in the array was removed, then what used to be at index 1 is now at index 0. React will be confused because it already had a key called 0 and the reconciliation process will not work correctly. This is why we tend to use database IDs instead. Chances are your list isn't as simple as the one from above and it comes from a database so this shouldn't be a problem.

**The first argument is an object of "props"**

Just like HTML elements can be assigned attributes like **id** or **title**, a React element can also receive a list of attributes when it gets rendered. The **Button** element received a **label** attribute. In React, the list of attributes received by a React element is known as **props**. A React function component receives this list as its first argument. The list is passed as an object with keys representing the attribute names, and values representing the values assigned to them.

When using a function component, you don’t have to name the object holding the list of attributes as **props**, but that is the standard practice. When using class components, which we will do in the following example, the same list of attributes is always presented with a special instance property named **props**.

*Receiving props is optional. Some components will not have any props. However, a component’s return value is not optional. A React component cannot return "undefined" (either explicitly or implicitly)—it has to return a value. It can return "null" to tell the renderer to ignore its output.*

You can use object destructuring whenever you use component props (or state, really). For example:

function Car(props) {

  return <h2>I am a {props.color} Car!</h2>;

}

This can be written as:

function Car({ color }) {

    return <h2>I am a {color} Car!</h2>;

}

This approach has many benefits, but the most important one is to visually inspect what props

are used in a component and make sure a component does not receive any extra props that are

not needed.

# Chapter 6: Hooks

Hooks were added to React in version 16.8. You must import Hooks from react.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed. Although Hooks generally replace class components, there are no plans to remove classes from React. Hooks will not work in React class components.

Hooks allow us to "hook" into React features such as state and lifecycle methods. For example a hook like useState is used to keep track of the application state. State generally refers to application data or properties that need to be tracked.

A Hook in a React component is a call to a special function. All Hook functions begin with the word **use**. The most widely-used Hook function in React is the **useState** one. You can use it to give a component stateful elements.

There are 3 rules for hooks:

* Hooks can only be called inside React function components.
* Hooks can only be called at the top level of a component.
* Hooks cannot be conditional

**Don’t call Hooks inside loops, conditions, or nested functions.** Instead, always use Hooks at the top level of your React function, before any early returns. By following this rule, you ensure that Hooks are called in the same order each time a component renders. That’s what allows React to correctly preserve the state of Hooks between multiple useState and useEffect calls.

**Only Call Hooks from React Functions. Don’t call Hooks from regular JavaScript functions.** Instead, you can:

* Call Hooks from React function components.
* Call Hooks from custom Hooks.

By following this rule, you ensure that all stateful logic in a component is clearly visible from its source code.

If you have stateful logic that needs to be reused in several components, you can build your own custom Hooks.

## useState

To see an example of that, let’s make a **Button** component respond to a click event. Let’s maintain the number of times it gets clicked in a **count** variable and display the value of that variable as the label of the button it renders.

const Button = () => {

  let count = 0;

  return (

    <button>{count}</button>

  );

};

ReactDOM.render(<Button />, mountNode);

This **count** variable will be the state element that we need to introduce to the example. It’s a piece of data that the UI will depend on (because we’re displaying it), and it is a state element because it is going to change over time.

***Every time you define a variable in your code, you will be introducing a state, and every time you change the value of that variable, you are mutating that state. The value stored in local state is lost by each render. Sin order to have a global store not changing through renders you must use “Context”. Keep that in mind.***

Before we can change the value of the count state, we need to learn about events.

### Responding to user events

You can add an event handler with an **onEvent** property (to the **button** element in this case). This could be an **onClick**, **onMouseOver**, **onScroll**, or **onSubmit**, among others.

For example, to make the program log a message to the console every time the button is clicked, we can do something like:

const Button = () => {

  let count = 0;

  return (

    <button onClick={() => console.log('Button clicked')}>

      {count}

    </button>

  );

};

ReactDOM.render(<Button />, mountNode);

Unlike the DOM version of the **onClick** attribute (which uses a string), React’s **onClick** attribute uses a function reference. You specify that inside curly brackets.

function func() { }

<button onClick={func} />

***Note:*** *Note how the event name is in camel case. All DOM-related attributes (which are handled by React) need to be in camel case, and React will display an error if they are not. React also supports using custom HTML attributes, and those have to be in all-lowercase format.*

*Some DOM attributes in React are slightly different from what they do in the regular DOM API. An example of that is the onChange event. In a regular browser, it’s usually fired when you click outside a form field (or tab out of it). In React, onChange fires whenever the value of a form field is changed (on every character added or removed).*

#### Managing State in React

Every React component can have its own **state**. What do we mean by *state*? The state is the **set of data that is managed by the component**.

Think about a form, for example. Each individual input element of the form is responsible for managing its state: what is written inside it. A button is responsible for knowing if it's being clicked, or not. If it's on focus. A link is responsible for knowing if the mouse is hovering it.

In React, or in any other components-based framework/library, all our applications are based and make heavy use of components state.

We manage state using the **useState** utility provided by React. It's technically a **hook**.

You import useState from React in this way:

import React, { useState } from 'react';

Calling useState() , you will get back a new state variable, an a function that we can call to alter its value.

useState() accepts the initial value of the state item and returns an array containing the state variable, and the function you call to alter the state.

This is important. We can't just alter the value of a state variable directly. We must call its modifier function. Otherwise the React component will not update its UI to reflect the changes of the data. Calling the modifier is the way we can tell React that the component state has changed.

The syntax is a bit weird, right? Since useState() returns an array we use array destructuring to access each individual item, like this:

const [count, setCount] = useState(0)

You can add as many calls you want, to useState()create as many state variables as you want:

const [count, setCount] = useState(0)

const [anotherCounter, setAnotherCounter] = useState

The argument is useState (“0”) assigns an initial value to the state variable **count**.

##### Reading and Updating State

To track state updates and trigger virtual DOM diffing and real DOM reconciliation, React needs to be aware of any changes that happen to any state elements that are used within components. To do this in an efficient way, React requires the use of special getters and setters for each state element you introduce in a component. This is where the **useState** Hook comes into play. It defines a state element and gives us a getter and setter for it.

Here’s what we need for the **count** state element we’re trying to implement:

const [count, setCount] = React.useState(0);

The **useState** function returns an array with two items. The first item is a value (getter), and the second item is a function (setter). I used array destructuring to give these items names. You can give them any names you want, but [**name, setName**] is the convention.

The first item "value" can be a string, number, array, or another type. In this case, we needed a number, and we needed to initialize that number with **0**. The argument to **React.useState** is used as the initial value of the state element.

The second item "function" will change the value of the state element (and trigger DOM processing if needed). Each time the **setCount** function is invoked, React will re-render the **Button** component, which will refresh all variables defined in the component (including the **count** value). The argument we pass to **setCount** becomes the new value for **count**. To update our state, we use our state updater function. We should never directly update state. Ex: count = 5 is not allowed.

The useState Hook can be used to keep track of strings, numbers, booleans, arrays, objects, and any combination of these!

To make the button increment its label, we need to invoke the **setCount** function within the **onClick** event and pass to it the current **count** value incremented by 1. Here’s the full code of the label-incrementing button example:

const Button = () => {

  const [count, setCount] = useState(0);

  return (

    <button onClick={() => setCount(count + 1)}>

      {count}

    </button>

  );

};

ReactDOM.render(<Button />, mountNode);

Go ahead and test that. The button will now increment its label on each click. If we had not made **count** a state variable, it’s value would have been lost between consecutive renders.

Note how we did not implement any actions to change the UI itself. We implemented an action to change a JavaScript object (in memory). Our UI implementation was basically telling React that we want the label of the button to always reflect the value of the **count** object. Our code didn’t do any DOM updates—React did.

Note also how I used the **const** keyword to define **count**, although it’s a value that gets changed. Our code will not change that value. React will when it uses a fresh call of the **Button** function to render the UI of its new state. In that fresh call, the **useState** function will give us a new fresh **count** value.

##### Updating Objects and Arrays in State

When state is updated, the entire state gets overwritten. What if we only want to update the color of our car? If we only called setCar({color: "blue"}), this would remove the brand, model, and year from our state.

We can use the JavaScript spread operator to help us.

Example: Use the JavaScript spread operator to update only one property of an object:

import { useState } from "react";

import ReactDOM from "react-dom";

function Car() {

  const [car, setCar] = useState({

    brand: "Ford",

    model: "Mustang",

    year: "1964",

    color: "red"

  });

  const updateColor = () => {

    setCar(previousState => {

      return { ...previousState, color: "blue" }

    });

  }

  return (

    <>

      <h1>My {car.brand}</h1>

      <p>

        It is a {car.color} {car.model} from {car.year}.

      </p>

      <button

        type="button"

        onClick={updateColor}

      >Blue</button>

    </>

  )

}

ReactDOM.render(<Car />, document.getElementById('root'));

## useCallback

The React useCallback Hook returns a memoized callback function.

Think of memoization as caching a value so that it does not need to be recalculated.

This allows us to isolate resource intensive functions so that they will not automatically run on every render.

The useCallback Hook only runs when one of its dependencies update.

This can improve performance.

The **useCallback** and **useMemo** Hooks are similar. The main difference is that useMemo returns a memoized *value* and useCallback returns a memoized *function*.

One reason to use useCallback is to prevent a component from re-rendering unless its props have changed. Pass an inline callback and an array of dependencies. useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

The syntax:

const memoizedCallback = useCallback(

  () => {

    doSomething(a, b);

  },

  [a, b],

);

In order to make use of the examples on the following two pages create an empty applications and just modify the App.js file accordingly. Index.js needs not to be touched.

import React, { useState, useCallback } from 'react'

const funccount = new Set();

const App = () => {

  const [count, setCount] = useState(0)

  const [number, setNumber] = useState(0)

  const incrementCounter = () => {

    setCount(count + 1)

  }

  const decrementCounter = () => {

    setCount(count - 1)

  }

  const incrementNumber = () => {

    setNumber(number + 1)

  }

  funccount.add(incrementCounter);

  funccount.add(decrementCounter);

  funccount.add(incrementNumber);

  alert(funccount.size);

  return (

    <div>

      Count: {count}

      <button onClick={incrementCounter}>

        Increase counter

      </button>

      <button onClick={decrementCounter}>

        Decrease Counter

      </button>

      <button onClick={incrementNumber}>

        increase number

      </button>

    </div>

  )

}

export default App;

**Without useCallback Hook:** The problem is that once the counter is updated, all three functions are recreated again. The alert increases by three at a time but if we update some states all the functions related to that states should only be re-instantiated. If another state value is unchanged, it should not be touched. Here, the filename is App.js.

import React, { useState, useCallback } from 'react'

var funccount = new Set();

const App = () => {

  const [count, setCount] = useState(0)

  const [number, setNumber] = useState(0)

  const incrementCounter = useCallback(() => {

    setCount(count + 1)

  }, [count])

  const decrementCounter = useCallback(() => {

    setCount(count - 1)

  }, [count])

  const incrementNumber = useCallback(() => {

    setNumber(number + 1)

  }, [number])

  funccount.add(incrementCounter);

  funccount.add(decrementCounter);

  funccount.add(incrementNumber);

  alert(funccount.size);

  return (

    <div>

      Count: {count}

      <button onClick={incrementCounter}>

        Increase counter

      </button>

      <button onClick={decrementCounter}>

        Decrease Counter

      </button>

      <button onClick={incrementNumber}>

        increase number

      </button>

    </div>

  )

}

export default App;

**With useCallback hook:** To solve this problem we can use the useCallback hook. Here, the filename is App.js.

## useEffect

The useEffect Hook allows you to perform side tasks (effects) in your components. Some examples of side effects are: fetching data, directly updating the DOM, and timers. useEffect accepts two arguments, the second argument is optional.

**useEffect(<function>, <dependency>)**

useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect. If this is not what we want, there are several ways to control when side effects run. In that case we should always include the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array, which puts a condition on useEffect to run on every render.

Let’s clarify these concepts with examples. In this example we use setTimeout() to count 1 second after the initial render:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom";

function Timer() {

  const [count, setCount] = useState(0);

  useEffect(() => {

    setTimeout(() => {

      setCount((count) => count + 1);

    }, 1000);

  });

  return <h1>I've rendered {count} times!</h1>;

}

ReactDOM.render(<Timer />, document.getElementById('root'));

Run that code and observe how it’s updating the counter every 1 sec.

But wait! It keeps counting forever! Now suppose that we want it to count for a specified number of times. So, to fix this issue, let's only run this effect on the initial render:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom";

function Timer() {

  const [count, setCount] = useState(0);

  useEffect(() => {

    setTimeout(() => {

      setCount((count) => count + 1);

    }, 1000);

  }, []); // <- add empty brackets here

  return <h1>I've rendered {count} times!</h1>;

}

ReactDOM.render(<Timer />, document.getElementById('root'));

What we have done here is we passed the useEffect hook and empty array as its dependency parameter. In that case useEffect checks the data in the array, and runs only when the data in the array is updated. Here an empty array represents a dependency which never gets fullfilled, because there is nothing to be updated there! So the useEffect runs, checks its dependency and stops immediately.

If we had provided data in this array, the running of the useEffect woul depende on the change of that data. That is to say, if the count variable updates the effect will run, if not it will not run. Let’s see:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom";

function Counter() {

  const [count, setCount] = useState(0);

  const [calculation, setCalculation] = useState(0);

  useEffect(() => {

    setCalculation(() => count \* 2);

  }, [count]); // <- add the count variable here

  return (

    <>

      <p>Count: {count}</p>

      <button onClick={() => setCount((c) => c + 1)}>+</button>

      <p>Calculation: {calculation}</p>

    </>

  );

}

ReactDOM.render(<Counter />, document.getElementById('root'));

When run this code, you will see that whenever you click on the button it changes the value **count**, increments it by 2. And since the useEffect is dependent on the count, this triggers it to invoke the **setCalculation()** function, which returns the count value doubled.

Note: You can also include more than one element in the dependency array, which means multiple dependency. In that case the running od useEffect is dependent on ANY of the elements changing, not ALL. (Show this by adding a constant count2 in the dependency array).

## useContext

Before we dwell on useContext Hook, it would be helpful to first describe what React Context is.

### What is React Context

React Context is a way to manage state globally. It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

In a typical React application, data is passed top-down (parent to child) via props, but such usage can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree. Using context, we can avoid passing props through intermediate elements.

Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.

Context is primarily used when some data needs to be accessible by *many* components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.

**If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.**

This *inversion of control* can make your code cleaner in many cases by reducing the amount of props you need to pass through your application and giving more control to the root components. Such inversion, however, isn’t the right choice in every case; moving more complexity higher in the tree makes those higher-level components more complicated and forces the lower-level components to be more flexible than you may want.

You’re not limited to a single child for a component. You may pass multiple children, or even have multiple separate “slots” for children.

This pattern is sufficient for many cases when you need to decouple a child from its immediate parents. You can take it even further with render props if the child needs to communicate with the parent before rendering.

However, sometimes the same data needs to be accessible by many components in the tree, and at different nesting levels. Context lets you “broadcast” such data, and changes to it, to all components below. Common examples where using context might be simpler than the alternatives include managing the current locale, theme, or a data cache.

Now let’s present a problem we can face with **useState**, and the provide a solution with **useContext**.

**The Problem**

State should be held by the highest parent component in the stack that requires access to the state.

To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.

To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

import { useState } from "react";

import ReactDOM from "react-dom";

function Component1() {

  const [user, setUser] = useState("Jesse Hall");

  return (

    <>

      <h1>{`Hello ${user}!`}</h1>

      <Component2 user={user} />

    </>

  );

}

function Component2({ user }) {

  return (

    <>

      <h1>Component 2</h1>

      <Component3 user={user} />

    </>

  );

}

function Component3({ user }) {

  return (

    <>

      <h1>Component 3</h1>

      <h2>{`Hello ${user} again!`}</h2>

    </>

  );

}

ReactDOM.render(<Component1 />, document.getElementById("root"));

Even though component 2 did not need the state, they had to pass the state along so that it could reach component 3.

**The Solution**

The solution is to create context. To create context, you must Import createContext and initialize it:

import { useState, createContext } from "react";

import ReactDOM from "react-dom";

const UserContext = createContext()

Next we'll use the Context Provider to wrap the tree of components that need the state Context. Wrap child components in the Context Provider and supply the state value:

function Component1() {

  const [user, setUser] = useState("Jesse Hall");

  return (

    <UserContext.Provider value={user}>

      <h1>{`Hello ${user}!`}</h1>

      <Component2 user={user} />

    </UserContext.Provider>

  );

}

Now, all components in this tree will have access to the user Context. In order to use the Context in a child component, we need to access it using the **useContext** Hook.

First, include the useContext in the import statement:

import { useState, createContext, useContext } from "react";

Then you can access the user Context in all components:

function Component5() {

  const user = useContext(UserContext);

  return (

    <>

      <h1>Component 5</h1>

      <h2>{`Hello ${user} again!`}</h2>

    </>

  );

}

In the next page we provide the full code.

import { useState, createContext, useContext } from "react";

import ReactDOM from "react-dom";

const UserContext = createContext();

function Component1() {

  const [user, setUser] = useState("Jesse Hall");

  return (

    <UserContext.Provider value={user}>

      <h1>{`Hello ${user}!`}</h1>

      <Component2 user={user} />

    </UserContext.Provider>

  );

}

function Component2() {

  return (

    <>

      <h1>Component 2</h1>

      <Component3 />

    </>

  );

}

function Component3() {

  return (

    <>

      <h1>Component 3</h1>

      <h2>{`Hello ${user} again!`}</h2>

    </>

  );

}

ReactDOM.render(<Component1 />, document.getElementById("root"));

## useReducer

The **useReducer** Hook is similar to the **useState** Hook. It allows for custom state logic. If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.

The useReducer Hook accepts two arguments: useReducer(<reducer>, <initialState>)

The reducer function contains your custom state logic and the initialState can be a simple value but generally will contain an object.

The useReducer Hook returns the current state and a dispatch method.

“Reducer” is a fancy word for a function that takes 2 or more values and returns 1 value. useReducer is in some way similar to the Array reduce function because it takes the same arguments, and basically works the same way. You pass a reducer function and an initial value (initial state). Your reducer receives the current **state** and an **action**, and returns the new state. We could write one that works just like the summation reducer:

useReducer((state, action) => {

  return state + action;

}, 0);

So… what triggers this function to run? How does the action get in there? Good questions.

useReducer returns an array of 2 elements, similar to the useState hook. The first is the current state, and the second is a dispatch function. Here’s how it looks in practice:

const [sum, dispatch] = useReducer((state, action) => {

  return state + action;

}, 0);

We’re using ES6 destructuring syntax to pull out the 2 values from the array and name them **sum** and **dispatch** here.

Important to notice, the “state” can be any kind of value. It doesn’t have to be an object. It could be a number, or an array, or anything else.

Let’s look at a complete useReducer example of a component using it to increment a number:

import React, { useReducer } from 'react';

function Counter() {

  // First render will create the state, and it will

  // persist through future renders

  const [sum, dispatch] = useReducer((state, action) => {

    return state + action;

  }, 0);

  return (

    <>

      {sum}

      <button onClick={() => dispatch(1)}>

        Add 1

      </button>

    </>

  );

}

You can see how clicking the button dispatches an action with a value of 1, which gets added to the current state, and then the component re-renders with the new (larger!) state.

I’m intentionally showing an example where the “action” doesn’t have the form { type: "INCREMENT\_BY", value: 1 } or some other such thing, because the reducers you create don’t *have to* follow the typical patterns from Redux. The world of Hooks is a new world: it’s worth considering whether you find old patterns valuable and want to keep them, or whether you’d rather change things up.

## useRef

The useRef Hook allows you to persist values between renders.

* It can be used to store a mutable value that does not cause a re-render when updated.
* It can be used to access a DOM element directly.
* It can also be used to track state changes.

### Does Not Cause Re-renders

If we tried to count how many times our application renders using the useState Hook, we would be caught in an infinite loop since this Hook itself causes a re-render.

To avoid this, we can use the useRef Hook.

import { useState, useEffect, useRef } from "react";

import ReactDOM from "react-dom";

function App() {

  const [inputValue, setInputValue] = useState("");

  const count = useRef(0);

  useEffect(() => {

    count.current = count.current + 1;

  });

  return (

    <>

      <input

        type="text"

        value={inputValue}

        onChange={(e) => setInputValue(e.target.value)}

      />

      <h1>Render Count: {count.current}</h1>

    </>

  );

}

ReactDOM.render(<App />, document.getElementById('root'));

useRef() only returns one item. It returns an Object called current. When we initialize useRef we set the initial value: useRef(0). It's like doing this:

const count = {current: 0}. We can access the count by using count.current.

Run this on your computer and try typing in the input to see the application render count increase.

### Accessing DOM Elements

In general, we want to let React handle all DOM manipulation. But there are some instances where useRef can be used without causing issues.

In React, we can add a ref attribute to an element to access it directly in the DOM.

Example: Use useRef to focus the input:

import { useRef } from "react";

import ReactDOM from "react-dom";

function App() {

  const inputElement = useRef();

  const focusInput = () => {

    inputElement.current.focus();

  };

  return (

    <>

      <input type="text" ref={inputElement} />

      <button onClick={focusInput}>Focus Input</button>

    </>

  );

}

ReactDOM.render(<App />, document.getElementById('root'));

### Tracking State Changes

The useRef Hook can also be used to keep track of previous state values.

This is because we are able to persist useRef values between renders.

Example: Use useRef to keep track of previous state values:

import { useState, useEffect, useRef } from "react";

import ReactDOM from "react-dom";

function App() {

  const [inputValue, setInputValue] = useState("");

  const previousInputValue = useRef("");

  useEffect(() => {

    previousInputValue.current = inputValue;

  }, [inputValue]);

  return (

    <>

      <input

        type="text"

        value={inputValue}

        onChange={(e) => setInputValue(e.target.value)}

      />

      <h2>Current Value: {inputValue}</h2>

      <h2>Previous Value: {previousInputValue.current}</h2>

    </>

  );

}

ReactDOM.render(<App />, document.getElementById('root'));

This time we use a combination of useState, useEffect, and useRef to keep track of the previous state.

In the useEffect, we are updating the useRef current value each time the inputValue is updated by entering text into the input field.

## useMemo

The React useMemo Hook returns a memoized value. Think of memoization as caching a value so that it does not need to be recalculated. In Computer Science, memoization is a concept used in general when we don’t need to recompute the function with a given argument for the next time as it returns the cached result. A memoized function remembers the results of output for a given set of inputs. For example, if there is a function to add two numbers, and we give the parameter as 1 and 2 for the first time the function will add these two numbers and return 3, but if the same inputs come again then we will return the cached value i.e. 3 and not compute with the add function again. In react also, we use this concept, whenever in the React component, the state and props do not change the component and the component does not re-render, it shows the same output. The useMemo hook is used to improve performance in our React application.

The useMemo Hook only runs when one of its dependencies update. This can improve performance.

The useMemo and useCallback Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function.

It’s syntax is:

const memoizedValue = useMemo(functionThatReturnsValue,  arrayDepencies)

In order to make use of the examples on the following two pages create an empty applications and just modify the App.js file accordingly. Index.js needs not to be touched.

import React, { useState } from 'react';

function App() {

  const [number, setNumber] = useState(0)

  const squaredNum = squareNum(number);

  const [counter, setCounter] = useState(0);

  // Change the state to the input

  const onChangeHandler = (e) => {

    setNumber(e.target.value);

  }

  // Increases the counter by 1

  const counterHander = () => {

    setCounter(counter + 1);

  }

  return (

    <div className="App">

      <h1>Welcome to Geeksforgeeks</h1>

      <input type="number" placeholder="Enter a number"

        value={number} onChange={onChangeHandler}>

      </input>

      <div>OUTPUT: {squaredNum}</div>

      <button onClick={counterHander}>Counter ++</button>

      <div>Counter : {counter}</div>

    </div>

  );

}

// function to square the value

function squareNum(number) {

  console.log("Squaring will be done!");

  return Math.pow(number, 2);

}

export default App;

**Output:**In the above example, we have an App component and this component is doing two things one is squaring a number on the given input and incrementing the counter. We have two states here number and counter, whenever any of the states change the component re-renders. For example, if we change the input value of the number the function squareNum runs, and if increment the counter again the function squareNum runs.

In this case, we can see that even if we changed the input number once, but clicked on-increment counter multiple times our function squareNum got executed whenever we clicked the increment counter button multiple times. This is happening because the App component re-renders whenever we change the state of the counter.

Now let’s solve this problem using the useMemo hook.

**Example:**When we use useMemo Hook

import React, { useState } from 'react';

function App() {

  const [number, setNumber] = useState(0)

  // Using useMemo

  const squaredNum = useMemo(() => {

    return squareNum(number);

  }, [number])

  const [counter, setCounter] = useState(0);

  // Change the state to the input

  const onChangeHandler = (e) => {

    setNumber(e.target.value);

  }

  // Increases the counter by 1

  const counterHander = () => {

    setCounter(counter + 1);

  }

  return (

    <div className="App">

      <h1>Welcome to Geeksforgeeks</h1>

      <input type="number" placeholder="Enter a number"

        value={number} onChange={onChangeHandler}>

      </input>

      <div>OUTPUT: {squaredNum}</div>

      <button onClick={counterHander}>Counter ++</button>

      <div>Counter : {counter}</div>

    </div>

  );

}

// function to square the value

function squareNum(number) {

  console.log("Squaring will be done!");

  return Math.pow(number, 2);

}

export default App;

**Output:**Now in the above example, we have used the user memo hook, here the function that returns the value i.e squareNum is passed inside the useMemo and inside the array dependencies, we have used the number as the squareNum will run only when the number changes. If we increase the counter and the number remains the same in the input field the squareNum doesn’t run again. Let’s see the output below.

## Custom Hooks

Hooks are reusable functions.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

Custom Hooks start with "use". Example: useFetch.

### Build a Hook

In the following code, we are fetching data in our Home component and displaying it.

We will use the [JSONPlaceholder](https://jsonplaceholder.typicode.com/) service to fetch fake data. This service is great for testing applications when there is no existing data. To learn more, check out the [JavaScript Fetch API](https://www.w3schools.com/js/js_api_fetch.asp) section.

Use the JSONPlaceholder service to fetch fake "todo" items and display the titles on the page:

Example: Index.js:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom";

const Home = () => {

  const [data, setData] = useState(null);

  useEffect(() => {

    fetch("https://jsonplaceholder.typicode.com/todos")

      .then((res) => res.json())

      .then((data) => setData(data));

  }, []);

  return (

    <>

      {data &&

        data.map((item) => {

          return <p key={item.id}>{item.title}</p>;

        })}

    </>

  );

};

ReactDOM.render(<Home />, document.getElementById("root"));

The fetch logic may be needed in other components as well, so we will extract that into a custom Hook.

Move the fetch logic to a new file to be used as a custom Hook:

Example: useFetch.js:

import { useState, useEffect } from "react";

const useFetch = (url) => {

  const [data, setData] = useState(null);

  useEffect(() => {

    fetch(url)

      .then((res) => res.json())

      .then((data) => setData(data));

  }, [url]);

  return [data];

};

export default useFetch;

Now the new Index.js will be:

import ReactDOM from "react-dom";

import useFetch from "./useFetch";

const Home = () => {

  const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");

  return (

    <>

      {data &&

        data.map((item) => {

          return <p key={item.id}>{item.title}</p>;

        })}

    </>

  );

};

ReactDOM.render(<Home />, document.getElementById("root"));

We have created a new file called useFetch.js containing a function called useFetch which contains all of the logic needed to fetch our data.

We removed the hard-coded URL and replaced it with a url variable that can be passed to the custom Hook.

Lastly, we are returning our data from our Hook.

In index.js, we are importing our useFetch Hook and utilizing it like any other Hook. This is where we pass in the URL to fetch data from.

Now we can reuse this custom Hook in any component to fetch data from any URL.

# Chapter 7: Miscellaneous Topics

## React useContext and useReducer Hooks

Instead of using Redux as state management. We can use the inbuilt hooks that are available in React itself. Eventually, you can replace or move the project that is dependent on Redux to the inbuilt hooks.

The useContext hook is used to create common data that can be accessed throughout the component hierarchy without passing the props down manually to each level. Context defined will be available to all the child components without involving “props”. React Context is a powerful state management feature in React. Instead of passing the props down through each component, React Context allows you to broadcast props to the components below.

The useReducer hook is used for complex state manipulations and state transitions. … useReducer is a React hook function that accepts a reducer function, and an initial state. const [state, dispatch] = useReducer(reducer, initialState);This hook function returns an array with 2 values.

I am using the usual use case of the Todo List example for easy understanding.

**Step 1: Initial State and Actions**

//Initial State and Actions

const initialState = {

    todoList: []

};

const actions = {

    ADD\_TODO\_ITEM: "ADD\_TODO\_ITEM",

    REMOVE\_TODO\_ITEM: "REMOVE\_TODO\_ITEM",

    TOGGLE\_COMPLETED: "TOGGLE\_COMPLETED"

};

**Step 2: Reducers to Handle Actions**

/Reducer to Handle Actions

const reducer = (state, action) => {

    switch (action.type) {

        case actions.ADD\_TODO\_ITEM:

            return {

                todoList: [

                    ...state.todoList,

                    {

                        id: new Date().valueOf(),

                        label: action.todoItemLabel,

                        completed: false

                    }

                ]

            };

        case actions.REMOVE\_TODO\_ITEM: {

            const filteredTodoItem = state.todoList.filter(

                (todoItem) => todoItem.id !== action.todoItemId

            );

            return { todoList: filteredTodoItem };

        }

        case actions.TOGGLE\_COMPLETED: {

            const updatedTodoList = state.todoList.map((todoItem) =>

                todoItem.id === action.todoItemId

                    ? { ...todoItem, completed: !todoItem.completed }

                    : todoItem

            );

            return { todoList: updatedTodoList };

        }

        default:

            return state;

    }

};

Breakdown of the Code: We use the usual Switch Case Statements to Evaluate the Actions.

* First Case ADD\_TODO\_ITEM -action spread the existing list and add a new todo item to the list with id(unique-ish), label(user-entered value), and completed flag.
* Second Case REMOVE\_TODO\_ITEM -action filter out the to-do item that needs to be removed based on the id.
* Third Case TOGGLE\_COMPLETED - action loop through all the to-do items and toggle the completed flag based on the id.

**Step 3: Create the Context and Provider to Dispatch the Actions.**

//Context and Provider

const TodoListContext = React.createContext();

const Provider = ({ children }) => {

    const [state, dispatch] = React.useReducer(reducer, initialState);

    const value = {

        todoList: state.todoList,

        addTodoItem: (todoItemLabel) => {

            dispatch({ type: actions.ADD\_TODO\_ITEM, todoItemLabel });

        },

        removeTodoItem: (todoItemId) => {

            dispatch({ type: actions.REMOVE\_TODO\_ITEM, todoItemId });

        },

        markAsCompleted: (todoItemId) => {

            dispatch({ type: actions.TOGGLE\_COMPLETED, todoItemId });

        }

    };

    return (

        <TodoListContext.Provider value={value}>children}TodoListContext.Provider>

    );

};

In this step, we create the TodoListContext and a Provider function that returns the TodoListContext’s Provider.

Here is the Breakdown of the Code.

* Here we pass the reducer function and theinitialState to the useReducer hook. This will return state and dispatch. The state will have the initialState. And the dispatch is used to trigger our actions, just like in redux.
* In the value object, we have todoList state, and three functions addTodoItem, removeTodoItem, and markAsCompleted which trigger ADD\_TODO\_ITEM, REMOVE\_TODO\_ITEM, and TOGGLE\_COMPLETED actions respectively.
* We pass the value object as a prop to the TodoListContext's Provider, so that we can access it using useContext.

**Step 4: Create the Two Components that will use the store.  
AddTodo & TodoList**

// AddTodo Component with Input field and Add Button

const AddTodo = () => {

    const [inputValue, setInputValue] = React.useState("");

    const { addTodoItem } = React.useContext(TodoListContext);

    return (

        <>

            <input

                type="text"

                value={inputValue}

                placeholder={"Type and add todo item"}

                onChange={(e) => setInputValue(e.target.value)}

            />

            <button

                onClick={() => {

                    addTodoItem(inputValue);

                    setInputValue("");

                }}

            >

                Add

            </button>

        </>

    );

};

In thisAddTodocomponent, we use the useContext to subscribe to our TodoListContext and getting addTodoItem dispatch function.

//TodoList Component to show the list

const TodoList = () => {

    const { todoList, removeTodoItem, markAsCompleted } = React.useContext(

        TodoListContext

    );

    return (

        <ul>

            {todoList.map((todoItem) => (

                <li

                    className={`todoItem ${todoItem.completed ? "completed" : ""}`}

                    key={todoItem.id}

                    onClick={() => markAsCompleted(todoItem.id)}

                >

                    {todoItem.label}

                    <button

                        className="delete"

                        onClick={() => removeTodoItem(todoItem.id)}

                    >

                        X

                    </button>

                </li>

            ))}

        </ul>

    );

};

In TodoList component, we are using useContext to subscribe to the TodoListContext and getting the todoList state, removeTodoItemand andmarkAsCompleted dispatch functions. We are mapping through todoList and rendering the to-do items and a remove(X) button next to them. On clicking on an item we are marking it as complete and when clicking on X the button we are removing it from the list.

**Step 5: Final step, wrapping the above two components to the Provider.**

//Final Wrapper

export default function App() {

    return (

        <Provider>

            <AddTodo />

            <TodoList />

        </Provider>

    );

}

## A Guide to React Context and useContext() Hook

The React context provides data to components no matter how deep they are in the components tree. The context is used to manage global data, e.g. global state, theme, services, user settings, and more.

In this post, you'll learn how to use the context concept in React.

### How to use the context

Using the context in React requires 3 simple steps: *creating* the context, *providing* the context, and *consuming* the context.

**A. Creating the context**

The built-in factory function createContext(default) creates a context instance:

import { createContext } from 'react';

const Context = createContext('Default Value');

The factory function accepts one optional argument: the default value.

**B. Providing the context**

Context.Provider component available on the context instance is used to provide the context to its child components, no matter how deep they are.

To set the value of context use the value prop available on the <Context.Provider value={value} />:

function Main() {

    const value = 'My Context Value';

    return (

        <Context.Provider value={value}>

            <MyComponent />

        </Context.Provider>

    );

}

Again, what's important here is that all the components that'd like later to consume the context have to be wrapped inside the provider component.

If you want to change the context value, simply update the value prop.

**C. Consuming the context**

Consuming the context can be performed in 2 ways.

The first way, the one I recommend, is to use the useContext(Context) React hook:

import { useContext } from 'react';

function MyComponent() {

    const value = useContext(Context);

    return <span>{value}</span>;

}

The hook returns the value of the context: value = useContext(Context). The hook also makes sure to re-render the component when the context value changes.

The second way is by using a render function supplied as a child to Context.Consumer special component available on the context instance:

function MyComponent() {

    return (

        <Context.Consumer>

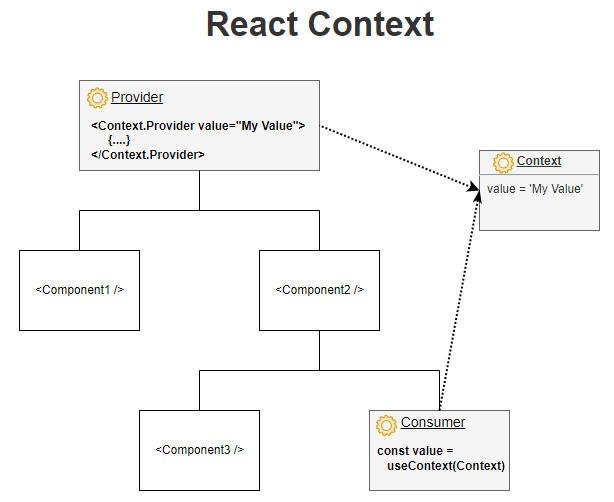
            {value => <span>{value}</span>}

        </Context.Consumer>

    );

}

Again, in case if the context value changes, <Context.Consumer> will re-render its render function.



You can have as many consumers as you want for a single context. If the context value changes (by changing the value prop of the provider <Context.Provider value={value} />), then all consumers are immediately notified and re-rendered.

If the consumer isn't wrapped inside the provider, but still tries to access the context value (using useContext(Context) or <Context.Consumer>), then the value of the context would be the default value argument supplied to createContext(defaultValue) factory function that had created the context.

### When do you need context?

The main idea of using the context is to allow your components to access some global data and re-render when that global data is changed. Context solves the props drilling problem: when you have to pass down props from parents to children.

You can hold inside the context:

* global state
* theme
* application configuration
* authenticated user name
* user settings
* preferred language
* a collection of services

On the other side, you should think carefully before deciding to use context in your application.

First, integrating the context adds complexity. Creating the context, wrapping everything in the provider, using the useContext() in every consumer — this increases complexity.

Secondly, adding context makes it more difficult to unit test the components. During unit testing, you would have to wrap the consumer components into a context provider. Including the components that are indirectly affected by the context — the ancestors of context consumers!

### Use case: global user name

The simplest way to pass data from a parent to a child component is when the parent assigns props to its child component:

function Application() {

    const userName = "John Smith";

    return <UserInfo userName={userName} />;

}

function UserInfo({ userName }) {

    return <span>{userName}</span>;

}

The parent component <Application /> assigns userName data to its child component <UserInfo name={userName} /> using the userName prop.

That's the usual way how data is passed using props. You can use this approach without problems.

The situation changes when <UserInfo /> child component isn't a direct child of <Application /> but is contained within multiple ancestors.

For example, let's say that <Application /> component (the one having the global data userName) renders <Layout /> component, which in turn renders <Header /> component, which in turn finally renders <UserInfo /> component (that'd like to access userName).

Here's how such a structuring would look:

function Application() {

    const userName = "John Smith";

    return (

        <Layout userName={userName}>

            Main content

        </Layout>

    );

}

function Layout({ children, userName }) {

    return (

        <div>

            <Header userName={userName} />

            <main>

                {children}

            </main>

        </div>

    )

}

function Header({ userName }) {

    return (

        <header>

            <UserInfo userName={userName} />

        </header>

    );

}

function UserInfo({ userName }) {

    return <span>{userName}</span>;

}

You can see the problem: because <UserInfo /> component renders deep down in the tree, and all the parent components (<Layout /> and <Header />) have to pass the userName prop.

This problem is also known as [props drilling](https://kentcdodds.com/blog/prop-drilling).

React context is a possible solution. Let's see how to apply it in the next section.

#### Context to the rescue

As a quick reminder, applying the React context requires 3 actors: the context, the provider extracted from the context, and the consumer.

Here's how the sample application would look when applying the context to it:

import { useContext, createContext } from 'react';

const UserContext = createContext('Unknown');

function Application() {

    const userName = "John Smith";

    return (

        <UserContext.Provider value={userName}>

            <Layout>

                Main content

            </Layout>

        </UserContext.Provider>

    );

}

function Layout({ children }) {

    return (

        <div>

            <Header />

            <main>

                {children}

            </main>

        </div>

    );

}

function Header() {

    return (

        <header>

            <UserInfo />

        </header>

    );

}

function UserInfo() {

    const userName = useContext(UserContext);

    return <span>{userName}</span>;

}

Let's look into more detail what has been done.

First, const UserContext = createContext('Unknown') creates the context that's going to hold the user name information.

Second, inside the <Application /> component, the application's child components are wrapped inside the user context provider: <UserContext.Provider value={userName}>. Note that the value prop of the provider component is important: this is how you set the value of the context.

Finally, <UserInfo /> becomes the consumer of the context by using the built-in useContext(UserContext) hook. The hook is called with the context as an argument and returns the user name value.

<Layout /> and <Header /> intermediate components don't have to pass down the userName prop. That is the great benefit of the context: it removes the burden of passing down data through the intermediate components.

#### When context changes

When the context value is changed by altering value prop of the context provider (<Context.Provider value={value} />), then all of its consumers are being notified and re-rendered.

For example, if I change the user name from 'John Smith' to 'Smith, John Smith', then <UserInfo /> consumer immediately re-renders to display the latest context value:

import { createContext, useEffect, useState } from 'react';

const UserContext = createContext('Unknown');

function Application() {

    const [userName, setUserName] = useState('John Smith');

    useEffect(() => {

        setTimeout(() => {

            setUserName('Smith, John Smith');

        }, 2000);

    }, []);

    return (

        <UserContext.Provider value={userName}>

            <Layout>

                Main content

            </Layout>

        </UserContext.Provider>

    );

}

// ...

Open the [demo](https://codesandbox.io/s/react-context-example-change-hw32y?file=/src/Application.js) and you'd see 'John Smith' (context value) displayed on the screen. After 2 seconds, the context value changes to 'Smith, John Smith', and correspondingly the screen is updated with the new value.

The demo shows that <UserInfo /> component, the consumer that renders the context value on the screen, re-renders when the context value changes.

function UserInfo() {

    const userName = useContext(UserContext);

    return <span>{userName}</span>;

}

### Updating the context

The React Context API is stateless by default and doesn't provide a dedicated method to update the context value from consumer components.

But this can be easily implemented by integrating a state management mechanism (like useState() or useReducer() hooks), and providing an update function right in the context next to the value itself.

In the following example, <Application /> component uses useState() hook to manage the context value.

import { createContext, useState, useContext, useMemo } from 'react';

const UserContext = createContext({

    userName: '',

    setUserName: () => { },

});

function Application() {

    const [userName, setUserName] = useState('John Smith');

    const value = useMemo(

        () => ({ userName, setUserName }),

        [userName]

    );

    return (

        <UserContext.Provider value={value}>

            <UserNameInput />

            <UserInfo />

        </UserContext.Provider>

    );

}

function UserNameInput() {

    const { userName, setUserName } = useContext(UserContext);

    const changeHandler = event => setUserName(event.target.value);

    return (

        <input

            type="text"

            value={userName}

            onChange={changeHandler}

        />

    );

}

function UserInfo() {

    const { userName } = useContext(UserContext);

    return <span>{userName}</span>;

}

<UserNameInput /> consumer reads the context value, from where userName and setUserName are extracted. The consumer then can update the context value by invoking the update function setUserName(newContextValue).

<UserInfo /> is another consumer of the context. When <UserNameInput /> updates the context, this component is updated too.

Note that <Application /> memoizes the context value. Memoization keeps the context value object the same as long as userName is the same, preventing re-rendering of consumers every time the <Application /> re-renders.

Otherwise, without memoization, const value = { userName, setUserName } would create different object instances during re-rendering of <Application />, triggering re-rendering in context consumers. See more about [referential equality of objects](https://dmitripavlutin.com/how-to-compare-objects-in-javascript/#1-referential-equality).

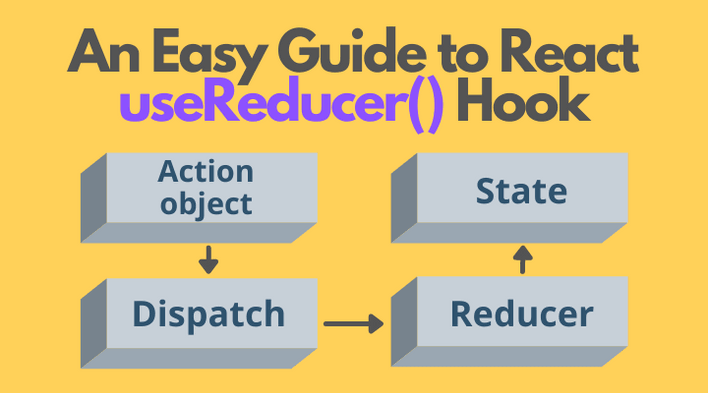
### Conclusion

The context in React is a concept that lets you supply child components with global data, no matter how deep they are in the components tree.

Using the context requires 3 steps: creating, providing, and consuming the context.

When integrating the context into your application, consider that it adds a good amount of complexity. Sometimes drilling the props through 2-3 levels in the hierarchy isn't a big problem.

## An Easy Guide to React useReducer() Hook



If you've used useState() hook to manage non-trivial state like a list of items, where you need to add, update and remove items in the state, you might have noticed that the state management logic takes a good part of the component body.

That's a problem because the React component in nature should contain the logic that calculates the output. But the state management logic is a different concern that should be managed in a separate place. Otherwise, you get a mix of state management and rendering logic in one place, and that's difficult to read, maintain, and test!

To help you separate the concerns (rendering and state management) React provides the hook useReducer(). The hook does so by extracting the state management out of the component.

Let's see how the useReducer() hook works. As a nice bonus, you will find in the post a real-world example that greatly helps undersanding how reducers work.

### useReducer()

The useReducer(reducer, initialState) hook accept 2 arguments: the *reducer* function and the *initial state*. The hook then returns an array of 2 items: the current state and the *dispatch* function.

import { useReducer } from 'react';

function MyComponent() {

    const [state, dispatch] = useReducer(reducer, initialState);

    const action = {

        type: 'ActionType'

    };

    return (

        <button onClick={() => dispatch(action)}>

            Click me

        </button>

    );

}

Now, let's decipher what the terms of *initial state*, *action object*, *dispatch*, and *reducer* mean.

**A. Initial state**

The *initial state* is the value the state is initialized with.

For example, in the case of a counter state, the initial value could be:

// initial state

const initialState = {

    counter: 0

};

**B. Action object**

An *action object* is an object that describes how to update the state.

Typically, the action object would have a property type — a string describing what kind of state update the reducer must do.

For example, an action object to increase the counter can look as follows:

const action = {

    type: 'increase'

};

user is a property that holds the information about the user to add.

**C. Dispatch function**

The *dispatch* is a special function that dispatches an action object.

The dispatch function is created for your by the useReducer() hook:

const [state, dispatch] = useReducer(reducer, initialState);a

Whenever you want to update the state (usually from an event handler or after completing a fetch request), you simply call the dispatch function with the appropritate action object: dispatch(actionObject).

**D. Reducer function**

The *reducer* is a pure function that accepts 2 parameters: the *current state* and an *action object*. Depending on the action object, the reducer function must update the state in an immutable manner, and return the new state.

The following reducer function supports the increase and decrease of a counter state:

function reducer(state, action) {

    let newState;

    switch (action.type) {

        case 'increase':

            newState = { counter: state.counter + 1 };

            break;

        case 'descrease':

            newState = { counter: state.counter - 1 };

            break;

        default:

            throw new Error();

    }

    return newState;

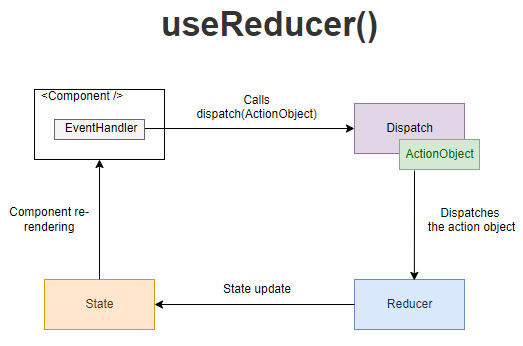
}

The reducer above doesn't modify directly the current state in the state variable, but rather creates a new state object stored in newState, then returns it.

React checks the difference between the new and the current state to determine whether the state has been updated, so do not mutate the current state directly.

**E. Wiring everything**

Wiring all these terms together, here's how the state update using a reducer works.



As a result of an event handler or after completing a fetch request, you call the *dispatch* function with the *action object*.

Then React redirects the action object and the current state value to the *reducer* function.

The reducer function uses the action object and performs a state update, returning the new state.

React then checks whether the new state differs from the previous one. If the state has been updated, React re-renders the component and useReducer() returns the new state value: [newState, ...] = useReducer(...).

Note that useReducer() design is based on the [Flux architecture](https://facebook.github.io/flux/docs/in-depth-overview).

If all these terms sound too abstract, then you have the right feeling! Let's see how useReducer() works in an interesting example.

### Implementing a stopwatch

The task is to implement a stopwatch. The stopwatch has 3 buttons: Start, Stop and Reset, and has a number displaying the passed seconds.

Now let's think about structuring the state of the stopwatch.

There are 2 important state properties: a boolean indicating whether the stopwatch runs (let's name it isRunning) and a number indicating the number of passed seconds (let's name it time). As result, here's how the *initial state* can look like:

const initialState = {

    isRunning: false,

    time: 0

};

The initial state indicates that the stopwatch starts as inactive and at 0 seconds.

Then let's consider what *action objects* the stopwatch should have. It's easy to find that we need 4 kinds of actions: to start, stop and reset the stopwatch running process, as well as tick the time each second.

// The start action object

{ type: 'start' }

// The stop action object

{ type: 'stop' }

// The reset action object

{ type: 'reset' }

// The tick action object

{ type: 'tick' }

Having the state structure, as well as the possible actions, let's use the *reducer* function to define how the action objects update the state:

function reducer(state, action) {

    switch (action.type) {

        case 'start':

            return { ...state, isRunning: true };

        case 'stop':

            return { ...state, isRunning: false };

        case 'reset':

            return { isRunning: false, time: 0 };

        case 'tick':

            return { ...state, time: state.time + 1 };

        default:

            throw new Error();

    }

}

Finally, here's the component Stopwatch that wires everything together by invoking the useReducer() hook:

import { useReducer, useEffect, useRef } from 'react';

function Stopwatch() {

    const [state, dispatch] = useReducer(reducer, initialState);

    const idRef = useRef(0);

    useEffect(() => {

        if (!state.isRunning) {

            return;

        }

        idRef.current = setInterval(() => dispatch({ type: 'tick' }), 1000);

        return () => {

            clearInterval(idRef.current);

            idRef.current = 0;

        };

    }, [state.isRunning]);

    return (

        <div>

            {state.time}s

            <button onClick={() => dispatch({ type: 'start' })}>

                Start

            </button>

            <button onClick={() => dispatch({ type: 'stop' })}>

                Stop

            </button>

            <button onClick={() => dispatch({ type: 'reset' })}>

                Reset

            </button>

        </div>

    );

}

The click event handlers of the Start, Stop, and Reset buttons correspondingly use the dispatch() function to dispatch the necessary action object.

Inside the useEffect() callback, if state.isRunning is true, the setInterval() timer function dispatches the tick action object each second dispatch({type: 'tick'}).

Each time the reducer() function updates the state, the component re-renders as a result and receives the new state.

### Reducer mental model

To solidify your knowledge even more, let's see a real-world example that works similarly to a reducer.

Imagine you're the captain of a ship in the first half of the 20th century. The captain's bridge has a special communication device called *engine order telegraph* (see the picture above). This communication tool is used to transmit commands from the bridge to the engine room. Typical commands would be to move *back slowly*, move *ahead half* power, *stop*, etc.

You're on the bridge and the ship is at full stop. You (the captain) want the ship to move forward at full speed. You'd approach the engine order telegraph and set the handle to *ahead full*. The engineers in the engine room, having the same device, see the *ahead full* command, and set the engine to the corresponding regime.

The *engine order telegraph* is the *dispatch* function, the *commands* are the *action objects*, the *engineers in the engine room* are the *reducer* function, and the *engine regime* is the *state*.

The engine order telegraph helps separate the bridge from the engine room. The same way the useReducer() hook helps separate the rendering from the state management logic.

### Conclusion

The useReducer() hook in React lets you separate the state management from the rendering logic of the component.

const [state, dispatch] = useReducer(reducer, initialState) accepts 2 argument: the reducer function and the initial state. Also, the reducer returns an array of 2 items: the current state and the dispatch function.

When you'd like to update the state, simply call dispatch(action) with the appropriate action object. The action object is then forwarded to the reducer() function that updates the state. If the state has been updated by the reducer, then the component re-renders, and [state, ...] = useReducer(...) hook returns the new state value.

useReducer() fits great with relatively complex state update (requiring at least 2-3 update actions). For simple state management, simply use useState().

*Challenge: write a custom hook myUseState() that works exactly useState(), only that it uses the useReducer() hook to manage the state. Write your solution in a comment below!*

## React Events

Just like HTML DOM events, React can perform actions based on user events.

React has the same events as HTML: click, change, mouseover etc.

**Adding Events**

React events are written in camelCase syntax:

**onClick** instead of **onclick**.

React event handlers are written inside curly braces:

**onClick={shoot}** instead of **onClick="shoot()"**.

React:

<button onClick={shoot}>Take the Shot!</button>

HTML:

<button onclick="shoot()">Take the Shot!</button>

Example: Put the shoot function inside the Football component:

import React from 'react';

import ReactDOM from 'react-dom';

function Football() {

    const shoot = () => {

        alert("Great Shot!");

    }

    return (

        <button onClick={shoot}>Take the shot!</button>

    );

}

ReactDOM.render(<Football />, document.getElementById('root'));

**Passing Arguments**

To pass an argument to an event handler, use an arrow function.

Example: Send "Goal!" as a parameter to the shoot function, using arrow function:

import React from 'react';

import ReactDOM from 'react-dom';

function Football() {

    const shoot = (a) => {

        alert(a);

    }

    return (

        <button onClick={() => shoot("Goal!")}>Take the shot!</button>

    );

}

ReactDOM.render(<Football />, document.getElementById('root'));

**React Event Object**

Event handlers have access to the React event that triggered the function. In our example the event is the "click" event.

Example: Arrow Function: Sending the event object manually:

import React from 'react';

import ReactDOM from 'react-dom';

f

unction Football() {

    const shoot = (a, b) => {

        alert(b.type);

        /\*

        'b' represents the React event that triggered the function,

        in this case the 'click' event

        \*/

    }

    return (

        <button onClick={(event) => shoot("Goal!", event)}>

Take the shot!

</button>

    );

}

ReactDOM.render(<Football />, document.getElementById('root'));

## React Conditional Rendering

In React, you can conditionally render components. There are several ways to do this.

**if Statement**

We can use the if JavaScript operator to decide which component to render.

Example: We'll use these two components:

function MissedGoal() {

    return <h1>MISSED!</h1>;

}

function MadeGoal() {

    return <h1>Goal!</h1>;

}

Example: Now, we'll create another component that chooses which component to render based on a condition:

import React from 'react';

import ReactDOM from 'react-dom';

function MissedGoal() {

    return <h1>MISSED!</h1>;

}

function MadeGoal() {

    return <h1>GOAL!</h1>;

}

function Goal(props) {

    const isGoal = props.isGoal;

    if (isGoal) {

        return <MadeGoal />;

    }

    return <MissedGoal />;

}

ReactDOM.render(

    <Goal isGoal={false} />,

    document.getElementById('root')

);

Try changing the **isGoal** attribute to true and run it again.

**Logical && Operator**

Another way to conditionally render a React component is by using the && operator.

Example: We can embed JavaScript expressions in JSX by using curly braces:

import React from 'react';

import ReactDOM from 'react-dom';

function Garage(props) {

    const cars = props.cars;

    return (

        <>

            <h1>Garage</h1>

            {cars.length > 0 &&

                <h2>

                    You have {cars.length} cars in your garage.

                </h2>

            }

        </>

    );

}

const cars = ['Ford', 'BMW', 'Audi'];

ReactDOM.render(

    <Garage cars={cars} />,

    document.getElementById('root')

);

If cars.length is equates to true, the expression after && will render.

Try emptying the cars array:

const cars = [];

ReactDOM.render(

    <Garage cars={cars} />,

    document.getElementById('root')

);

**Ternary Operator**

Another way to conditionally render elements is by using a ternary operator.

condition ? true : false

We will go back to the goal example.

Example: Return the MadeGoal component if isGoal is true, otherwise return the MissedGoal component:

import React from 'react';

import ReactDOM from 'react-dom';

function MissedGoal() {

    return <h1>MISSED!</h1>;

}

function MadeGoal() {

    return <h1>GOAL!</h1>;

}

function Goal(props) {

    const isGoal = props.isGoal;

    return (

        <>

            {isGoal ? <MadeGoal /> : <MissedGoal />}

        </>

    );

}

ReactDOM.render(

    <Goal isGoal={false} />,

    document.getElementById('root')

);

## React Lists

In React, you will render lists with some type of loop. The JavaScript map() array method is generally the preferred method.

Example: Let's render all of the cars from our garage:

import React from 'react';

import ReactDOM from 'react-dom';

function Car(props) {

    return <li>I am a {props.brand}</li>;

}

function Garage() {

    const cars = ['Ford', 'BMW', 'Audi'];

    return (

        <>

            <h1>Who lives in my garage?</h1>

            <ul>

                {cars.map((car) => <Car brand={car} />)}

            </ul>

        </>

    );

}

ReactDOM.render(<Garage />, document.getElementById('root'));

If you run this example in your create-react-app, you will receive a warning that there is no "key" provided for the list items.

**Keys**

Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list.

Keys need to be unique to each sibling. But they can be duplicated globally. Generally, the key should be a unique ID assigned to each item. As a last resort, you can use the array index as a key. But we don’t recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state.

Example: Let's refactor our previous example to include keys:

import React from 'react';

import ReactDOM from 'react-dom';

function Car(props) {

    return <li>I am a {props.brand}</li>;

}

function Garage() {

    const cars = [

        { id: 1, brand: 'Ford' },

        { id: 2, brand: 'BMW' },

        { id: 3, brand: 'Audi' }

    ];

    return (

        <>

            <h1>Who lives in my garage?</h1>

            <ul>

                {cars.map((car) => <Car key={car.id} brand={car.brand} />)}

            </ul>

        </>

    );

}

ReactDOM.render(<Garage />, document.getElementById('root'));

**Extracting Components with Keys**

Keys only make sense in the context of the surrounding array.

For example, if you [extract](https://reactjs.org/docs/components-and-props.html#extracting-components) a ListItem component, you should keep the key on the <ListItem /> elements in the array rather than on the <li> element in the ListItem itself.

function ListItem(props) {

    const value = props.value;

    return (

        // Wrong! There is no need to specify the key here:

        <li key={value.toString()}>

            {value}

        </li>

    );

}

function NumberList(props) {

    const numbers = props.numbers;

    const listItems = numbers.map((number) =>

        // Wrong! The key should have been specified here:

        <ListItem value={number} />

    );

    return (

        <ul>

            {listItems}

        </ul>

    );

}

const numbers = [1, 2, 3, 4, 5];

ReactDOM.render(

    <NumberList numbers={numbers} />,

    document.getElementById('root')

);

The above code has incorrect key usage. The correct usage should be as below:

function ListItem(props) {

    // Correct! There is no need to specify the key here:

    return <li>{props.value}</li>;

}

function NumberList(props) {

    const numbers = props.numbers;

    const listItems = numbers.map((number) =>

        // Correct! Key should be specified inside the array.

        <ListItem key={number.toString()} value={number} />

    );

    return (

        <ul>

            {listItems}

        </ul>

    );

}

const numbers = [1, 2, 3, 4, 5];

ReactDOM.render(

    <NumberList numbers={numbers} />,

    document.getElementById('root')

);

A good rule of thumb is that elements inside the map() call need keys.

## React Forms

Just like in HTML, React uses forms to allow users to interact with the web page.

**Adding Forms in React**

You add a form with React like any other element:

Example: Add a form that allows users to enter their name:

mport React from 'react';

import ReactDOM from 'react-dom';

function MyForm() {

    return (

        <form>

            <label>Enter your name:

                <input type="text" />

            </label>

        </form>

    )

}

ReactDOM.render(<MyForm />, document.getEl

This will work as normal, the form will submit and the page will refresh. But this is generally not what we want to happen in React. We want to prevent this default behavior and let React control the form.

**Handling Forms**

Handling forms is about how you handle the data when it changes value or gets submitted.

In HTML, form data is usually handled by the DOM. In React, form data is usually handled by the components.

When the data is handled by the components, all the data is stored in the component state.

You can control changes by adding event handlers in the onChange attribute. We can use the useState Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

Example: Use the onChange event to manage the input:

import { useState } from "react";

import ReactDOM from 'react-dom';

function MyForm() {

    const [name, setName] = useState("");

    return (

        <form>

            <label>Enter your name:

                <input

                    type="text"

                    value={name}

                    onChange={(e) => setName(e.target.value)}

                />

            </label>

        </form>

    )

}

ReactDOM.render(<MyForm />, document.getElementById('root'));

**Submitting Forms**

You can control the submit action by adding an event handler in the onSubmit attribute for the <form>:

Example: Add a submit button and an event handler in the onSubmit attribute:

import { useState } from "react";

import ReactDOM from 'react-dom';

function MyForm() {

    const [name, setName] = useState("");

    const handleSubmit = (event) => {

        event.preventDefault();

        alert(`The name you entered was: ${name}`)

    }

    return (

        <form onSubmit={handleSubmit}>

            <label>Enter your name:

                <input

                    type="text"

                    value={name}

                    onChange={(e) => setName(e.target.value)}

                />

            </label>

            <input type="submit" />

        </form>

    )

}

ReactDOM.render(<MyForm />, document.getElementById('root'));

**Multiple Input Fields**

You can control the values of more than one input field by adding a name attribute to each element.

We will initialize our state with an empty object.

To access the fields in the event handler use the **event.target.name** and **event.target.value** syntax.

To update the state, use square brackets [bracket notation] around the property name.

Example: Write a form with two input fields:

import { useState } from "react";

import ReactDOM from "react-dom";

function MyForm() {

    const [inputs, setInputs] = useState({});

    const handleChange = (event) => {

        const name = event.target.name;

        const value = event.target.value;

        setInputs(values => ({ ...values, [name]: value }))

    }

    const handleSubmit = (event) => {

        event.preventDefault();

        alert(inputs);

    }

    return (

        <form onSubmit={handleSubmit}>

            <label>Enter your name:

                <input

                    type="text"

                    name="username"

                    value={inputs.username || ""}

                    onChange={handleChange}

                />

            </label>

            <label>Enter your age:

                <input

                    type="number"

                    name="age"

                    value={inputs.age || ""}

                    onChange={handleChange}

                />

            </label>

            <input type="submit" />

        </form>

    )

}

ReactDOM.render(<MyForm />, document.getElementById('root'));

**Note:** We use the same event handler function for both input fields, we could write one event handler for each, but this gives us much cleaner code and is the preferred way in React.

**Textarea**

The textarea element in React is slightly different from ordinary HTML.

In HTML the value of a textarea was the text between the start tag <textarea> and the end tag </textarea>.

<textarea>

    Content of the textarea.

</textarea>

In React the value of a textarea is placed in a value attribute. We'll use the useState Hook to mange the value of the textarea:

Example: A simple textarea with some content:

import { useState } from "react";

import ReactDOM from "react-dom";

function MyForm() {

    const [textarea, setTextarea] = useState(

        "The content of a textarea goes in the value attribute"

    );

    const handleChange = (event) => {

        setTextarea(event.target.value)

    }

    return (

        <form>

            <textarea value={textarea} onChange={handleChange} />

        </form>

    )

}

ReactDOM.render(<MyForm />, document.getElementById('root'));

**Select**

A drop down list, or a select box, in React is also a bit different from HTML. In HTML, the selected value in the drop down list was defined with the selected attribute:

HTML:

<select>

    <option value="Ford">Ford</option>

    <option value="Volvo" selected>Volvo</option>

    <option value="Fiat">Fiat</option>

</select>

In React, the selected value is defined with a value attribute on the select tag:

Example: A simple select box, where the selected value "Volvo" is initialized in the constructor:

import { useState } from "react";

import ReactDOM from "react-dom";

function MyForm() {

    const [myCar, setMyCar] = useState("Volvo");

    const handleChange = (event) => {

        setMyCar(event.target.value)

    }

    return (

        <form>

            <select value={myCar} onChange={handleChange}>

                <option value="Ford">Ford</option>

                <option value="Volvo">Volvo</option>

                <option value="Fiat">Fiat</option>

            </select>

        </form>

    )

}

ReactDOM.render(<MyForm />, document.getElementById('root'));

By making these slight changes to <textarea> and <select>, React is able to handle all input elements in the same way.

## React Router

Create React App doesn't include page routing. React Router is the most popular solution.

**Add React Router**

To add React Router in your application, run this in the terminal from the root directory of the application:

npm i -D react-router-dom

***Note:****This text uses React Router v6. If you are upgrading from v5, you will need to use the @latest flag:*

npm i -D react-router-dom@latest

**Folder Structure**

To create an application with multiple page routes, let's first start with the file structure. Within the src folder, we'll create a folder named pages with several files:

src\pages\:

* Layout.js
* Home.js
* Blogs.js
* Contact.js
* NoPage.js

Each file will contain a very basic React component.

**Basic Usage**

Now we will use our Router in our index.js file.

Example: Use React Router to route to pages based on URL:

import ReactDOM from "react-dom";

import { BrowserRouter, Routes, Route } from "react-router-dom";

import Layout from "./pages/Layout";

import Home from "./pages/Home";

import Blogs from "./pages/Blogs";

import Contact from "./pages/Contact";

import NoPage from "./pages/NoPage";

export default function App() {

    return (

        <BrowserRouter>

            <Routes>

                <Route path="/" element={<Layout />}>

                    <Route index element={<Home />} />

                    <Route path="blogs" element={<Blogs />} />

                    <Route path="contact" element={<Contact />} />

                    <Route path="\*" element={<NoPage />} />

                </Route>

            </Routes>

        </BrowserRouter>

    );

}

ReactDOM.render(<App />, document.getElementById("root"));

The above file lists the contents of index.js.

We wrap our content first with <BrowserRouter>.

Then we define our <Routes>. An application can have multiple <Routes>. Our basic example only uses one.

<Route>s can be nested. The first <Route> has a path of / and renders the Layout component.

The nested <Route>s inherit and add to the parent route. So the blogs path is combined with the parent and becomes /blogs.

The Home component route does not have a path but has an index attribute. That specifies this route as the default route for the parent route, which is /.

Setting the path to \* will act as a catch-all for any undefined URLs. This is great for a 404 error page.

**Pages / Components**

The Layout component has <Outlet> and <Link> elements.

The <Outlet> renders the current route selected.

<Link> is used to set the URL and keep track of browsing history.

Anytime we link to an internal path, we will use <Link> instead of <a href="">.

The "layout route" is a shared component that inserts common content on all pages, such as a navigation menu.

import { Outlet, Link } from "react-router-dom";

const Layout = () => {

    return (

        <>

            <nav>

                <ul>

                    <li>

                        <Link to="/">Home</Link>

                    </li>

                    <li>

                        <Link to="/blogs">Blogs</Link>

                    </li>

                    <li>

                        <Link to="/contact">Contact</Link>

                    </li>

                </ul>

            </nav>

            <Outlet />

        </>

    )

};

export default Layout;

The above file lists the contents of Layout.js.

const Home = () => {

    return <h1>Home</h1>;

};

export default Home;

The above file lists the contents of Home.js.

const Blogs = () => {

    return <h1>Blog Articles</h1>;

};

export default Blogs;

The above file lists the contents of Blogs.js.

const Contact = () => {

    return <h1>Contact Me</h1>;

};

export default Contact;

The above file lists the contents of Contact.js.

const NoPage = () => {

    return <h1>404</h1>;

};

export default NoPage;

The above file lists the contents of NoPage.js.

# Appendix 1: Use of Curly Brackets in React

Curly brackets have been used extensively in this book in different situations and in different contexts. This might be confusing to the newcomers of React. Therefore it will be very useful to cover thoroughly the use of curly brackets and also to mention where to use curly braces { } and when to use parenthesis ( ).

## How braces { } are used

Curly braces { } are special syntax in JSX. It is used to evaluate a JavaScript expression during compilation. A JavaScript expression can be a variable, function, an object, or any code that resolves into a value.

Let’s take an example.

* Evaluating a JavaScript variable

const yellowStyle = { color: 'yellow' }

  < Star style = { yellowStyle } />

Which is the same as:

<Star style={{color: 'yellow'}} />

* Evaluating a function or event handler

const doSomething = () => {

    // Function Scope

};

  render() {

    return (

      <div onClick={this.whoIsThis}>

        Hello World

      </div>

    )

  }

This is not be confused with Class methods in ES6, which also uses curly braces { }:

class Car {

  constructor(name) {

    this.brand = name;

  }

}

* When rendering a JSX element, that is in a render() function.

render() {

  return (

    <div style={{ color: Math.random() < 0.5 ? 'green' : 'red' }}>

      How do you like this?

    </div>

  );

}

But notice that ReactDOM.render() does not require curly brackets. Instead, it uses parenthesis like regular function and takes its arguments inside parenthesis:

ReactDOM.render(<HelloMessage message="Hello" />, document.getElementById("root"));

* When initializing state.

state = {

  visible: true;

}

* In a setState function.

\_handleClose = () => {

this.setState({ visible: false });

}

* When referencing props.

function Button(props) {

  //returns a Dom element here. Eg:

  return <button type="submit">{props.label}</button>;

}

render() {

  return (

    <button>{this.props.label}</button>

  );

}

* When defining objects.

const vehicleOne = {

    brand: 'Ford',

    model: 'Mustang',

    type: 'car',

    year: 2021,

    color: 'red'

}

* When destructuring objects (Note: Array destructuring is done by square brackets).

const { PI, E, SQRT2 } = Math;

This is the same as:

const PI = Math.PI;

const E = Math.E;

const SQRT2 = Math.SQRT2;

* In interpolated strings with backticks.

<div>

    {`Random value is: ${Math.random()}`}

</div>

* Although not a necessity, it’s conventional to surround single-line comments with curly brackets in React.

<script>

  {/\* run this function when the document is loaded \*/}

  window.onload = function() {

{/\*create a couple of elements in an empty HTML page\*/}

const heading = document.createElement("h1");

  const heading\_text = document.createTextNode("Big Head!");

  heading.appendChild(heading\_text);

  document.body.appendChild(heading);

}

</script>

* In an inline styling we need to use two curly braces, which are nested. Inline styles in React work the way you’d expect if you’ve used them in HTML. Just like in HTML, inline styles in React are set using the style attribute. Also, similarly to HTML, inline styles override the classes in React as well.

Now, let’s talk about differences. To style your components or elements in JSX, you must provide an object (or reference to an object) representing the styles. To make sure that JSX can decipher your style object, you must place it within a pair of curly braces.

style={{ color: 'green', fontSize: 16 }}

First pair declares that it is an attribute value, second pair defines an object as a value.

Note that this is the same as, where you define your style object beforehand:

const mystyle = { color: 'green', fontSize: 16 }

style={ mystyle }

## How parenthesis ( ) are used

* Functions use parenthesis, regardless of whether they take arguments or not:

function Hello() {

  console.log("Hello");

}

function HelloMessage({ message }) {

  console.log({ message });

}

ReactDOM.render(<HelloMessage message="Hello" />, document.getElementById("root"));

Arrow functions also use parenthesis. For example in the *\_handleClose* event handler above:

\_handleClose = () => {

this.setState({ visible: false });

}

One other thing about arrow functions is that if the function only has a single return line:

const square = (a) => {

    return a \* a;

};

You can make it even more concise by removing the curly brackets and the return keyword altogether.

const square = (a) => a \* a;

You can also remove the parentheses around the argument if the function receives a single argument:

const square = a => a \* a;

***Note****: See also Appendix2 items 7 and 8.*

* Parenthesis are used to group multiline of codes on JavaScript **return** statement so to prevent semicolon inserted automatically in the wrong place:

render() {

return (

    <div style={{ color: Math.random() < 0.5 ? 'green' : 'red' }}>

      How do you like this?

    </div>

  );

}

But if only a single line is returned, there is no need for parenthesis.

By the way, it is not necessary to add a semicolon in JavaScript. JavaScript engine automatically inserts a semicolon at the first possible opportunity on a line after a return statement. If the JavaScript engine places the semicolon where it should not be, your code won’t compile.

# Appendix 2: React – Commonly Faced Problems

This appendix provides detailed instructions for some common problems that a beginner React.js learner might make.

1. **Not Starting a Component Name with a Capital Letter:**

A React component must have a name which starts with a capital letter. If the component name does not start with a capital letter, the component use will be treated as a built-in element such as a div or span.

For example:

class greeting extends React.Component {

  // ...

}

If you try to render <greeting /> , React will ignore the above and you will get a warning:

[WARNING]

The tag <greeting> is unrecognized in this browser. If you meant to render a React component,

start its name with an uppercase letter.

The bigger problem here is when you decide to name your component button or img . React will ignore your component and just render a vanilla HTML button or img tag.

class button extends React.Component {

  render() {

    return <button>My button</button>

  }

}

ReactDom.render(<button />, mountNode);

Note how the "My Button" was not rendered above and React just rendered an empty HTML button element. React will not warn you in this case.

1. **Using Single Quotes Instead of Back-ticks:**

Strings created with back-ticks (` `) are different from strings created with single quotes (' '). We create a string using back-ticks when we need to include dynamic expressions inside that string (without resorting to string concatenation).

Let’s say you want a string that always reports the current time:

*"Time is … "*

// Current time string

const time = new Date().toLocaleTimeString();

// When using regular strings (single or double quotes),

// you need to use string concatenation:

'Time is ' + time;

// When using back-ticks,

// you can inject the time in the string using ${}

`Time is ${time}`;

Also, when using string literals (with back-ticks), you can create a string that spans multiple lines:

const template = `I

  CAN

  SPAN

  Multiple Lines`;

You can’t do that with regular strings.

1. **Using React.PropTypes:**

The **PropTypes** object was removed from React. It used to be available as **React.PropTypes** but you cannot use that anymore.

Instead, you need to:

* Add the new **prop-types** package to your project: npm install prop-types
* Import it: import PropTypes from 'prop-types'

Then you can use it. For example: PropTypes.string .

If you incorrectly use React.PropTypes , you will get errors like:

TypeError: Cannot read property 'string' of undefined

1. **Confusing Functions with Classes:**

Can you tell what’s wrong with the following code?

class Numbers extends React.Component {

  const arrayOfNumbers = \_.range(1, 10);

  // ...

}

The code above is invalid because inside the body of a JavaScript class you don’t have the freedom of doing just about anything. You can only define methods and properties using limited syntax.

This is a bit confusing because the {} used in the class syntax looks like the plain-old block scope, but it’s not.

On the other hand, inside a function-based component you DO have the freedom of doing just about anything:

// Totally Okay:

const Number = (props) => {

  const arrayOfNumbers = \_.range(1, 10);

  // ...

};

1. **Forgetting That Another App Instance is Still Using the Same Port:**

To run a web server, your need to use a host (like 127.0.0.1 ) and a port (like 8080 ) to make the server listen for request on a valid http address.

Once the web server is running successfully, it has control over that port. You cannot use the same port for anything else. The port will be busy.

If you try to run the same server in another terminal, you will get an error that the port is "in use". Something like:

Error: listen EADDRINUSE 127.0.0.1:8080

Be aware that sometimes a web server might be running in the background or inside a detached screen/tmux session. You don’t see it, but it is still occupying the port. To restart your server, you need to "kill" the one that’s still running.

To identify the process that’s using a certain port, you can either use a command like ps (and grep for something about your app) or if you know the port number you can use the lsof command:

lsof -i :8080

By the, let’s remind you that React uses the port 3000 by default.

1. **Forgetting to Create Environment Variables:**

Some projects depend on the existence of shell environment variables in order to start. If you run these projects without the needed environment variables, they will try to use undefined values for them and will potentially give you some cryptic errors.

For example, if a project connects to a database like MongoDB, chances are it uses an environment variable like process.env.MONGO\_URI to connect to it. This allows the project to be used with different MongoDB instances in different environments.

To locally run the project that connects to a MongoDB, you have to export a MONGO\_URI environment variable first. For example, if you have a local MongoDB running on port 27017, you would need to do this before running the project:

export MONGO\_URI = "mongodb://localhost:27017/mydb"

1. **Confusing Curly Brackets with Parentheses:**

Instead of:

return {

  something();

};

You need:

return (

  something();

);

The first one will try (and fail) to return an object while the second one will correctly call the something() function and return what that function returns.

Since any <tag> in JSX will translate to a function call, this problem applies when returning any JSX.

This problem is also common in arrow functions' short syntax.

Instead of:

const Greeting = () => {

  <div>

    Hello World

  </div>

};

You need:

const Greeting = () => (

  <div>

    Hello World

  </div>

);

1. **Not Wrapping Objects with Parentheses when necessary:**

Normally objects are surrounded with curly brackets. But doing that does not mean that we also have to use prenthesis when necessary. So instead of:

const myAction = () => { type: 'DO\_THIS' };

You need:

const myAction = () => ({ type: 'DO\_THIS' });

Not using parenthesis around the curly brackets makes JavaScript think you are starting a function scope. So without wrapping the object in parentheses, you would not be using the arrow function short syntax at all. You will actually be defining a label for a string inside a regular arrow function block scope.

Wrapping the curly brackets in parenthesis is your hint to JavaScript that you do not want to start a function scope but rather return an object (within the arrow function short syntax).

Also in function syntax instead of:

this.setState(prevState => { answer: 42 });

You need:

this.setState(prevState => ({ answer: 42 }));

1. **Not Using the Right Capitalization of API Methods and Properties:**

* It is **React.Component**, not **React.component**.
* It is **componentDidMount**, not **ComponentDidMount**.
* It’s usually **ReactDOM**, not **ReactDom** .
* If, instead of **props.userName** , you incorrectly use **props.username** or **props.UserName**, you will be using an undefined value. Pay attention to that

1. **Confusing the State Object with Instance Properties:**

In a class component, you can define a local state object and later access it with this:

class Greeting extends React.Component {

  state = {

    name: "World",

  };

  render() {

    return `Hello ${this.state.name}`;

  }

}

The above will output "Hello World".

You can also define other local instant properties beside state:

class Greeting extends React.Component {

  user =

    {

      name: "World",

    };

  render() {

    return `Hello ${this.user.name}`;

  }

}

The above will also output "Hello World".

The state instance property is a special one because React will manage it. You can only change it through setState and React will react when you do. However, all other instance properties that you define will have no effect on the rendering algorithm. You can change this.user in the above example as you wish and React will not trigger a render cycle in React.

1. **Assuming import/export Will Always Work**

The import/export feature is an official feature in JavaScript (since 2015). However, it’s the only ES2015 feature that is not yet fully supported in modern browsers and the latest Node.

The popular configuration of a React project uses Webpack and Babel. Both allow the use of this feature and compile it down to something all browsers can understand. You can only use import/export if you have something like Webpack or Babel in your flow.

However, having import/export in your React bundled app does not mean that you can just use them anywhere you want! For example, if you’re also doing server-side rendering through the latest Node, things will not work for you. You will most likely get an "unexpected token" error.

To have Node understand import/export as well (which is something you need if you use them on the front-end and you want to do SSR as well), you will have to run Node itself with a Babel preset (like the env preset) that can transpile them. You can use tools like pm2, nodemon, and babel-watch to do so in development and have Node restart every time you change something.

1. **Not Binding Handler Methods**

You can define class methods in a React component and then use them in the component’s render method. For example:

class Greeting extends React.Component {

  whoIsThis() {

    console.dir(this); // "this" is the caller of whoIsThis

    return "World";

  }

  render() {

    return `Hello ${this.whoIsThis()}`;

  }

}

ReactDOM.render(<Greeting />, mountNode);

I used the **whoIsThis** method inside the render method with **this** because inside render, the this keyword refers to the component instance associated with the DOM element that represents the component. React internally makes sure that “this” inside its class methods refers to the instance. However, JavaScript does not bind the instance automatically when you use a **reference** to the whoIsThis method.

The **console.dir** line in **whoIsThis** will correctly report the component instance because that method was called directly from the render method with an explicit caller (this). You should see the Greeting object in the console when you execute the code above.

Note: console.dir and console.log both print their arguments to the console, but console.log prints it in an HTML-like tree, whereas console.log prints it in a JSON-like tree.

However, when you use the same method in a delayed-execution channel, such as an event handler, the caller will no longer be explicit and the console.dir line will not report the component instance.

See the code and output (after clicking) below:

class Greeting extends React.Component {

  whoIsThis() {

    console.dir(this); // "this" is the caller of whoIsThis

    return "World";

  }

  render() {

    return (

      <div onClick={this.whoIsThis}>

        Hello World

      </div>

    )

  }

}

ReactDOM.render(<Greeting />, mountNode);

In the code above, React invokes the **whoIsThis** method when you click on the string, but it will not give you access to the component instance inside of it. This is why you get undefined when we click the string. This is a problem if your class method needs access to things like **this.props** and **this.state**. It will simply not work.

There are many solutions for this problem. You can wrap the method in an inline function or use the .**bind** call to force the method to remember its caller. Both are okay for infrequently updated components. You can also optimize the bind method by doing it in the **constructor** of the class instead of in the render method. However, the best solution to this method is to enable the ECMAScript class-fields feature (which is still stage-3) through Babel and just use an arrow function for the handlers:

class Greeting extends React.Component {

  whoIsThis = () => {

    console.dir(this);

  }

  render() {

    return (

      <div onClick={this.whoIsThis}>

        Hello World

      </div>

    );

  }

}

This will work as expected.