# let:

let allows us to define variables.Previously we declare our variables using “var” keyword. But it has some limitations when it comes to scope as it does not offer block scope.But now using “let” keyword we can declare truly block-scoped variables.

### var variables can be re-declared and updated

This means that we can do this within the same scope and won't get an error.

var greeter = "hey hi";

var greeter = "say Hello instead";

### let can be updated but not re-declared.

Just like var,  a variable declared with let can be updated within its scope. Unlike var, a let variable cannot be re-declared within its scope. So while this will work:

let greeting = "say Hi";

greeting = "say Hello instead";

# JavaScript Template Literals

## **Back-Tics Syntax**

**Template Literals** use back-ticks (``) rather than the quotes ("") to define a string:

### **Example**

let text = `Hello World!`;

With **template literals**, you can use both single and double quotes inside a string:

### **Example**

let text = `He's often called "Johnny"`;

**Template literals** allows multiline strings:

### **Example**

let text =  
`The quick  
brown fox  
jumps over  
the lazy dog`;

**Template literals** allow variables in strings:

### **Example**

let firstName = "John";  
let lastName = "Doe";  
  
let text = `Welcome ${firstName}, ${lastName}!`;

## the Rest Operator?

The **rest operator** is used to put the rest of some specific user-supplied values into a JavaScript array.

// Define a function with two regular parameters and one rest parameter:

function myBio(firstName, lastName, ...otherInfo) {

return otherInfo;

}

// Invoke myBio function while passing five arguments to its parameters:

myBio("Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male");

// The invocation above will return:

["CodeSweetly", "Web Developer", "Male"]

## the Spread Operator

Now, consider this example of a spread operator:

// Define a function with three parameters:

function myBio(firstName, lastName, company) {

return `${firstName} ${lastName} runs ${company}`;

}

// Use spread to expand an array’s items into individual arguments:

myBio(...["Oluwatobi", "Sofela", "CodeSweetly"]);

// The invocation above will return:

“Oluwatobi Sofela runs CodeSweetly”

The **spread operator** (...) helps you expand iterables into individual elements.

const myName = ["Sofela", "is", "my"];

const aboutMe = ["Oluwatobi", ...myName, "name."];

console.log(aboutMe);

// The invocation above will return:

[ "Oluwatobi", "Sofela", "is", "my", "name." ]

### Spread Example 2

const myName = "Oluwatobi Sofela";

console.log([...myName]);

// The invocation above will return:

[ "O", "l", "u", "w", "a", "t", "o", "b", "i", " ", "S", "o", "f", "e", "l", "a" ]

# JavaScript Arrow Function

Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax:

let myFunction = (a, b) => a \* b;

### **Before:**

hello = function() {  
  return "Hello World!";  
}

### **With Arrow Function:**

hello = () => {  
  return "Hello World!";  
}

If the function has only one statement, and the statement returns a value, you can remove the brackets *and* the return keyword:

### **Arrow Functions Return Value by Default:**

hello = () => "Hello World!";

If you have parameters, you pass them inside the parentheses:

### **Arrow Function With Parameters:**

hello = (val) => "Hello " + val;

In fact, if you have only one parameter, you can skip the parentheses as well:

### **Arrow Function Without Parentheses:**

hello = val => "Hello " + val;

## **Object-Oriented Programming Concepts**

* **Object** − An object is a real-time representation of any entity. According to Grady Brooch, every object is said to have 3 features −
  + **State** − Described by the attributes of an object.
  + **Behavior** − Describes how the object will act.
  + **Identity** − A unique value that distinguishes an object from a set of similar such objects.
* **Class** − A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.
* **Method** − Methods facilitate communication between objects.

### **Example: Declaring a class**

class Polygon {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

## **Creating Objects**

To create an instance of the class, use the new keyword followed by the class name. Following is the syntax for the same.

var object\_name= new class\_name([ arguments ])

### **Example: Instantiating a class**

var obj = new Polygon(10,12)

### **Example: Putting them together**

'use strict'

class Polygon {

constructor(height, width) {

this.h = height;

this.w = width;

}

test() {

console.log("The height of the polygon: ", this.h)

console.log("The width of the polygon: ",this. w)

}

}

//creating an instance

var polyObj = new Polygon(10,20);

polyObj.test();

## Callback functions

usefull when you want to run something after a specific task has been completed.

Assume that you are making two functions, one for storing data and one for reading data to and from server. Now writing operations takes 5 seconds and reading takes just 1. Let’s replicate the behaviour using setTimeout function.

Const students=[

{name= “Yash”, subject= “js”}, {name=”raj”,subject=”react”}

]

Function enrollStudent(student){

setTimeout(function(){

students.push(student);

console.log(“Added”)

},3000);

Function getStudents(){

setTimeout(function(){

let str=””;

students.foreach(function(student){

str+=`<li> ${student.name}</li>`

});

Document.getElementById(‘students’).innerHTML=str;

Console.log(“fetched”)}

,5000);

# Enhanced Object Literals

Creating objects in some languages can be expensive in terms of development time and processing power when a class must be declared before anything can be achieved. In JavaScript, it’s easy to create objects on the fly. For example:

// ES5-compatible code

var myObject = {

prop1: 'hello',

prop2: 'world',

output: function() {

console.log(this.prop1 + ' ' + this.prop2);

}

};

myObject.output(); // hello world

## ES6 Default Parameters

// Basic syntax

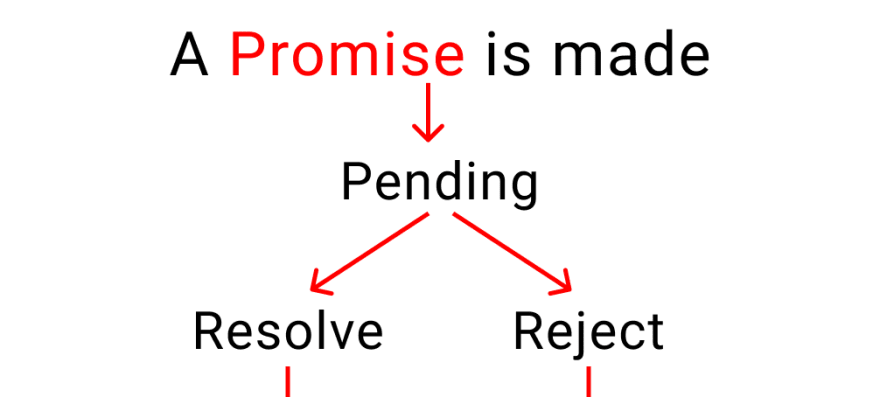
function multiply (a, b = 2) {

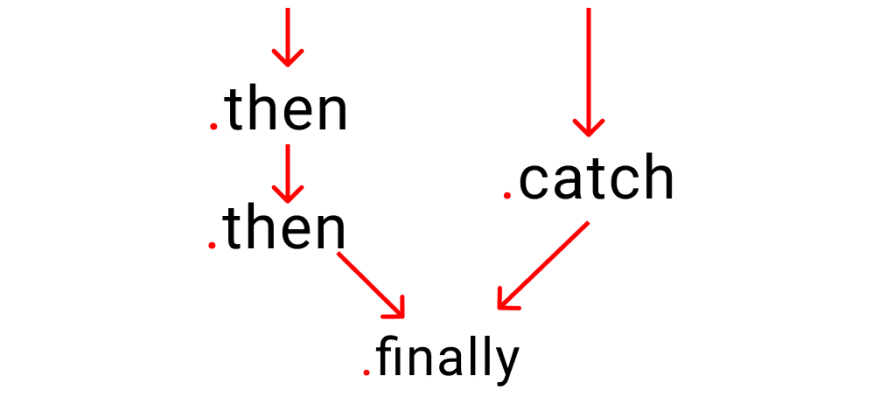
return a \* b;

}

multiply(5); // 10

# Promises





There're 3 states of a promise

* **Pending :** This is the initial stage. Nothing happens here. Think like this, your customer is taking his time to give an order. But, hasn't ordered anything.
* **Resolve :** This means that your customer has received his food and is happy.
* **Reject :** This means that your customer didn't receive his order and left the restaurant.

function func1(){

return new Promise(function(resolve,reject){

setTimeout(()=>{

const error = true;

if(!error){

console.log("Promise has been resolved");

resolve();

}

else{

console.log("Promise has not been resolved");

reject();

}

},2000);

})

}

func1().then(function(){

console.log("Thanks for resolving")

}).catch(function(){

console.log("Not fullfilled ")

})

# Async / AwaitBefore

To make a promise we wrote

function order(){

return new Promise( (resolve, reject) =>{

// Write code here

} )

}

## Now [ using Async / Await ]

In Async / Await method, we make promise like this

//👇 the magical keyword

async function order() {

// Write code here

}

### Async / Await -> try, catch

Here we work like this format

//👇 Magical keyword

async function kitchen(){

try{

// Let's create a fake problem

await abc;

}

catch(error){

console.log("abc does not exist", error)

}

finally{

console.log("Runs code anyways")

}

}

kitchen() // run the code

### A practical example

We don't know which topping customer prefers, Chocolate or peanuts?  
We need to stop our machine, go and ask our customer, "Sir, which topping would you love ?"

Notice here, only Our kitchen is stopped, but our staff outside the kitchen will still work like

* doing the dishes
* cleaning the tables
* taking orders, etc.
* Let's create a small promise, to ask which topping to use. The process takes 3 seconds.

function toppings\_choice (){

return new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve( console.log("which topping would you love?") )

},3000)

})

}

* now, let's create our kitchen function with the async keyword at first.

async function kitchen(){

console.log("A")

console.log("B")

console.log("C")

await toppings\_choice()

console.log("D")

console.log("E")

}

// Trigger the function

kitchen();

console.log("doing the dishes")

console.log("cleaning the tables")

console.log("taking orders")

What is React?

React is a **JavaScript** library created by **Facebook**

React is a **User Interface** (UI) library

React is a tool for building **UI components**

Environment Setup

Install Node.js

### **Step 1 - Create the Root Folder**

C:\Users\username\Desktop>mkdir reactApp

C:\Users\username\Desktop>cd reactApp

To create any module, it is required to generate the **package.json** file. Therefore, after Creating the folder, we need to create a **package.json** file. To do so you need to run the **npm init** command from the command prompt.

C:\Users\username\Desktop\reactApp>npm init -y

### **Step 2 - install React and react dom**

Since our main task is to install ReactJS, install it, and its dom packages, using **install react** and **react-dom** commands of npm respectively. You can add the packages we install, to **package.json** file using the **--save** option.

C:\Users\Tutorialspoint\Desktop\reactApp>npm install react --save

C:\Users\Tutorialspoint\Desktop\reactApp>npm install react-dom --save

### **Step 3 - Install webpack**

Since we are using webpack to generate bundler install webpack, webpack-dev-server and webpack-cli.

C:\Users\username\Desktop\reactApp>npm install webpack --save

C:\Users\username\Desktop\reactApp>npm install webpack-dev-server --save

C:\Users\username\Desktop\reactApp>npm install webpack-cli --save

### **Step 4 - Install babel**

Install babel, and its plugins babel-core, babel-loader, babel-preset-env, babel-preset-react and, html-webpack-plugin

C:\Users\username\Desktop\reactApp>npm install babel-core --save-dev

C:\Users\username\Desktop\reactApp>npm install babel-loader --save-dev

C:\Users\username\Desktop\reactApp>npm install babel-preset-env --save-dev

C:\Users\username\Desktop\reactApp>npm install babel-preset-react --save-dev

C:\Users\username\Desktop\reactApp>npm install html-webpack-plugin --save-dev

### **Step 5 - Create the Files**

To complete the installation, we need to create certain files namely, index.html, App.js, main.js, webpack.config.js and, .***babelrc***. You can create these files manually or, using **command prompt**.

C:\Users\username\Desktop\reactApp>type nul > index.html

C:\Users\username\Desktop\reactApp>type nul > App.js

C:\Users\username\Desktop\reactApp>type nul > main.js

C:\Users\username\Desktop\reactApp>type nul > webpack.config.js

C:\Users\username\Desktop\reactApp>type nul > .babelrc

### **Step 6 - Set Compiler, Server and Loaders**

Open **webpack-config.js** file and add the following code. We are setting webpack entry point to be main.js. Output path is the place where bundled app will be served. We are also setting the development server to **8001** port. You can choose any port you want.

**webpack.config.js**

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {

entry: './main.js',

output: {

path: path.join(\_\_dirname, '/bundle'),

filename: 'index\_bundle.js'

},

devServer: {

inline: true,

port: 8001

},

module: {

rules: [

{

test: /\.jsx?$/,

exclude: /node\_modules/,

loader: 'babel-loader',

query: {

presets: ['es2015', 'react']

}

}

]

},

plugins:[

new HtmlWebpackPlugin({

template: './index.html'

})

]

}

### **Step 7 - index.html**

This is just regular HTML. We are setting **div id = "app"** as a root element for our app and adding **index\_bundle.js** script, which is our bundled app file.

<!DOCTYPE html>

<html lang = "en">

<head>

<meta charset = "UTF-8">

<title>React App</title>

</head>

<body>

<div id = "app"></div>

<script src = 'index\_bundle.js'></script>

</body>

</html>

### **Step 8 − App.jsx and main.js**

This is the first React component. We will explain React components in depth in a subsequent chapter. This component will render **Hello World**.

**App.js**

import React, { Component } from 'react';

class App extends Component{

render(){

return(

<div>

<h1>Hello World</h1>

</div>

);

}

}

export default App;

**main.js**

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.js';

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

**Note** − Whenever you want to use something, you need to **import** it first. If you want to make the component usable in other parts of the app, you need to **export** it after creation and import it in the file where you want to use it.

Create a file with name **.babelrc** and copy the following content to it.

{

"presets":["env", "react"]

}

### **Step 9 - Running the Server**

The setup is complete and we can start the server by running the following command.

C:\Users\username\Desktop\reactApp>npm start

This was the manual way. More preferred way is :

## **Using the create-react-app command**

### **Step 1 - install create-react-app**

Browse through the desktop and install the Create React App using command prompt as shown below −

C:\Users\Tutorialspoint>cd C:\Users\Tutorialspoint\Desktop\

C:\Users\Tutorialspoint\Desktop>npx create-react-app my-app

### **Step 2: Run the project**

Finally, run the project using the start command.

npm start

#### Basic Folder Structure Explained

1. **package.json:**This File has the list of node dependencies which are needed.
2. **public/index.html:**When the application starts this is the first page that is loaded. This will be the only html file in the entire application since React is generally Written using **JSX**. Also, this file has a line of code **<div id=”root”></div>**. This line is very significant since all the application components are loade**d i**nto this div.
3. **src/index.js**: This is the javascript file corresponding to index.html. This file has the following line of code which is very significant.
4. **ReactDOM.render(<App />, document.getElementById(‘root’));**
5. The above line of code is telling that **App** Component has to be loaded into an html element with id **root**. This is nothing but the **div element** present in **index.html.**
6. **src/index.css**: The CSS file corresponding to index.js.
7. **src/App.js** : This is the file for **App** Component. **App** Component is the main component in React which acts as a container for all other components.
8. **src/App.css** : This is the CSS file corresponding to **App** Component
9. **build:** This is the folder where the built files are stored. React Apps can be developed using either JSX, or normal JavaScript itself, but using JSX definitely makes things easier to code for the developer :). But browsers do not understand JSX. So JSX needs to be converted into javascript before deploying. These converted files are stored in the build folder after bundling and minification. In order to see the build folder Run the following command

# React Render HTML

React's goal is in many ways to render HTML in a web page.

React renders HTML to the web page by using a function called ReactDOM.render().

## **The Render Function**

The ReactDOM.render() function takes two arguments, HTML code and an HTML element.

The purpose of the function is to display the specified HTML code inside the specified HTML element.

You'll notice a single <div> in the body of this file. This is where our React application will be rendered.

### **Example**

Display a paragraph inside an element with the id of "root":

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

The result is displayed in the <div id="root"> element:

<body>

<div id="root"></div>

</body>

Note that the element id does not have to be called "root", but this is the standard convention.

## **What is JSX?**

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

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

### **Example 1**

JSX:

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

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

### **Example 2**

Without JSX:

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

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

## **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. 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>;

## **Inserting a Large Block of HTML**

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

### **Example**

Create a list with three list items:

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: <></>.

## **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>;

## **Create Your First Component**

When creating a React component, the component's name *MUST* start with an upper case letter.

### **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>;

}

}

### **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 tutorial.

### **Example**

Create a Function component called Car

function Car() {

return <h2>Hi, I am a Car!</h2>;

}

## **Rendering a Component**

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'));

## **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'));

# React Props

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

props stands for properties.

## **React Props**

React Props are like function arguments in JavaScript *and* attributes in HTML.

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 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>;

}

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:

const carName = "Ford";

<Car brand={ carName } />

## **Objects as Props**

When you need to send an entire object or an array to the components as props, just create the object and then send it using any key.

So here is out Data object.

*const* people = [

    {

        id: 1,

        name: "Yash",

        city: "Ahmedabad"

    },

    {

        id: 2,

        name: "Raj",

        city: "Baroda"

    },

    {

        id: 3,

        name: "Peter",

        city: "Surat"

    }

]

Here is our Person component that renders individual personal details.

It receives props and in props it receives a single object of from out data.

*const* Person = (*props*) *=>* {

*const* { name, city } = *props*.details;

    return <div>

        <h1>{name}</h1>

        <p>{city}</p>

    </div>

}

Here is how to send the object to the component using props.

*const* NewComp = () *=>* {

    return <>{

        people.map((*person*) *=>* {

            return <*Person* details={*person*} key={*person*.id} />

        })

    }

    </>

}

Key is required as react needs a unique identifier for each item. Key can be anything as long as it is unique.

## **Children Props**

When you need to render an element inside your component then children prop is used. For example if you have created a component Person and you need to render an element inside the Person component which is in the parent component (Component where you are calling your component).

Taking the same data we will now send the city name of a person in p tag using children props

*const* NewComp = () *=>* {

    return <>{

        people.map((*person*) *=>* {

            return <*Person* name={*person*.name} key={*person*.id} ><p>{*person*.city}</p></*Person*>

        })

    }

    </>

}

*const* Person = (*props*) *=>* {

    return <div>

        <h1>{*props*.name}</h1>

        {*props*.children}

    </div>

}

## **Named Imports and Default Imports**

Named Imports : If your data is in some other file and you are exporting it without default keyword then it is called a named import. In named import you must use the same name in the file you are exporting.

For example this is out data.js file.

export *const* people = [

    {

        id: 1,

        name: "Yash",

        city: "Ahmedabad"

    },

    {

        id: 2,

        name: "Raj",

        city: "Baroda"

    },

    {

        id: 3,

        name: "Peter",

        city: "Surat"

    }

]

Export word makes the array available when imported in other file.

To import this we need to use the exact same name as we did when exporting it.

import { people } from './data';

The use of curly bracket around the name of the array is compulsory as it is a named import.

# React Events

## **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()".

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:**

function Football() {

const shoot = (a) => {

alert(a);

}

return (

<button onClick={() => shoot("Goal!")}>Take the shot!</button>

);

}

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

# React Conditional Rendering

Now, we'll create another component that chooses which component to render based on a condition:

function Goal(props) {

const isGoal = props.isGoal;

if (isGoal) {

return <MadeGoal/>;

}

return <MissedGoal/>;

}

ReactDOM.render(

<Goal isGoal={false} />,

document.getElementById('root')

);

MissedGoal Component

function MissedGoal() {

return <h1>MISSED!</h1>;

}

MadeGoal Component

function MadeGoal() {

return <h1>Goal!</h1>;

}

# React Lists

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'));

# React useState Hook

The React useState Hook allows us to track state in a function component.

## **Import useState**

To use the useState Hook, we first need to import it into our component.

### **Example:**

At the top of your component, import the useState Hook.

import { useState } from "react";

## **Initialize useState**

We initialize our state by calling useState in our function component.

useState accepts an initial state and returns two values:

* The current state.
* A function that updates the state.

### **Example:**

Initialize state at the top of the function component.

import { useState } from "react";

function FavoriteColor() {

const [color, setColor] = useState("");

}

Notice that again, we are destructuring the returned values from useState.

The first value, color, is our current state.

The second value, setColor, is the function that is used to update our state.

## **Update State**

To update our state, we use our state updater function.

We should never directly update state. Ex: color = "red" is not allowed.

### **Example:**

Use a button to update the state:

import { useState } from "react";

import ReactDOM from "react-dom";

function FavoriteColor() {

const [color, setColor] = useState("red");

return (

<>

<h1>My favorite color is {color}!</h1>

<button

type="button"

onClick={() => setColor("blue")}

>Blue</button>

</>

)

}

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

Create a single Hook that holds 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"

});

return (

<>

<h1>My {car.brand}</h1>

<p>

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

</p>

</>

)

}

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

## **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 the color of the car:

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'));

# React Forms

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.

You can control the submit action by adding an event handler in the onSubmit attribute for the <form>:

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.

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'));

# React useEffect Hooks

The useEffect Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM

It runs after every render.

*const* UseEffectBasics = () *=>* {

*const* [value, setValue] = useState(0);

  useEffect(() *=>* {

    console.log('call useEffect');

    if (value > 0) {

      document.title = `New Messages(${value})`;

    }

  });

  console.log('render component');

  return (

    <>

      <h1>{value}</h1>

      <button className='btn' onClick={() *=>* setValue(value + 1)}>

        click me

      </button>

    </>

  );

};

Now let’s add an event listened using react useEffect hook.

import React, { useState, useEffect } from 'react';

// cleanup function

// second argument

*const* UseEffectCleanup = () *=>* {

*const* [size, setSize] = useState(window.innerWidth);

*const* checkSize = () *=>* {

    setSize(window.innerWidth);

  };

  useEffect(() *=>* {

    console.log('useEffect');

    window.addEventListener('resize', checkSize);

  });

  console.log('render');

  return (

    <>

      <h1>window</h1>

      <h2>{size} PX</h2>

    </>

  );

};

Check the listeners in your browser. There will be multiple listeners added because a useEffect hook runs every time after a render. Now everytime a new addEventListener is executed it will add a new one. This is not a good thing, so we can make sure that useEffect runs only once in this case. That is why dependency list is used

useEffect(() *=>* {

    console.log('useEffect');

    window.addEventListener('resize', checkSize);

  }, []);

This empty array at the last part of useEffect is the dependency list. This useEffect will run only once.

# Other values as dependency

*const* [a, setA] = useState(0);

  useEffect(() *=>* {

    console.log("A is ", a);

  }, [a])

Here in useEffect state value “a” is as dependency list, so now the useEffect runs only when there is some change in the value of state variable “a”.

## **Effect Cleanup**

Some effects require cleanup to reduce memory leaks.

Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed.

We do this by including a return function at the end of the useEffect Hook.

### **Example:**

Clean up the timer at the end of the useEffect Hook:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom/client";

function Timer() {

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

useEffect(() => {

let timer = setTimeout(() => {

setCount((count) => count + 1);

}, 1000);

return () => clearTimeout(timer)

}, []);

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

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Timer />);

useReducer:

Used for state management.

Alternative to useState.

useState is built using useReducer.

| Scenario | useState | useReducer |
| --- | --- | --- |
| Type of State | Number,String Boolean | Object or Array |
| Number of state transition | One or two | Too many |
| Relate state transition | No | Yes |
| Local vs Global | Local | Global |

# React useContext Hook

## **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.

## **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, createContext, useContext } from "react";

import ReactDOM from "react-dom/client";

const UserContext = createContext();

function Component1() {

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

return (

<UserContext.Provider value={user}>

<h1>{`Hello ${user}!`}</h1>

<Component2 />

</UserContext.Provider>

);

}

function Component2() {

return (

<>

<h1>Component 2</h1>

<Component3 />

</>

);

}

function Component3() {

return (

<>

<h1>Component 3</h1>

<Component4 />

</>

);

}

function Component4() {

return (

<>

<h1>Component 4</h1>

<Component5 />

</>

);

}

function Component5() {

const user = useContext(UserContext);

return (

<>

<h1>Component 5</h1>

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

</>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Component1 />);

# React useRef Hook

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.

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

import ReactDOM from "react-dom/client";

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>

</>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

# React useCallback Hook

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.

### **Example:**

index.js

import { useState } from "react";

import ReactDOM from "react-dom/client";

import Todos from "./Todos";

const App = () => {

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

const [todos, setTodos] = useState([]);

const increment = () => {

setCount((c) => c + 1);

};

const addTodo = () => {

setTodos((t) => [...t, "New Todo"]);

};

return (

<>

<Todos todos={todos} addTodo={addTodo} />

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

</div>

</>

);

};

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

Todos.js

import { memo } from "react";

const Todos = ({ todos, addTodo }) => {

console.log("child render");

return (

<>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onClick={addTodo}>Add Todo</button>

</>

);

};

export default memo(Todos);

# React useMemo Hook

The React useMemo Hook returns a memoized value.

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

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

This can improve performance.

Performance example using the useMemo Hook:

import { useState, useMemo } from "react";

import ReactDOM from "react-dom/client";

const App = () => {

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

const [todos, setTodos] = useState([]);

const calculation = useMemo(() => expensiveCalculation(count), [count]);

const increment = () => {

setCount((c) => c + 1);

};

const addTodo = () => {

setTodos((t) => [...t, "New Todo"]);

};

return (

<div>

<div>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onClick={addTodo}>Add Todo</button>

</div>

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

<h2>Expensive Calculation</h2>

{calculation}

</div>

</div>

);

};

const expensiveCalculation = (num) => {

console.log("Calculating...");

for (let i = 0; i < 1000000000; i++) {

num += 1;

}

return num;

};

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

### Why Should I Use Redux?[​](https://redux.js.org/tutorials/essentials/part-1-overview-concepts#why-should-i-use-redux)

Redux helps you manage "global" state - state that is needed across many parts of your application.

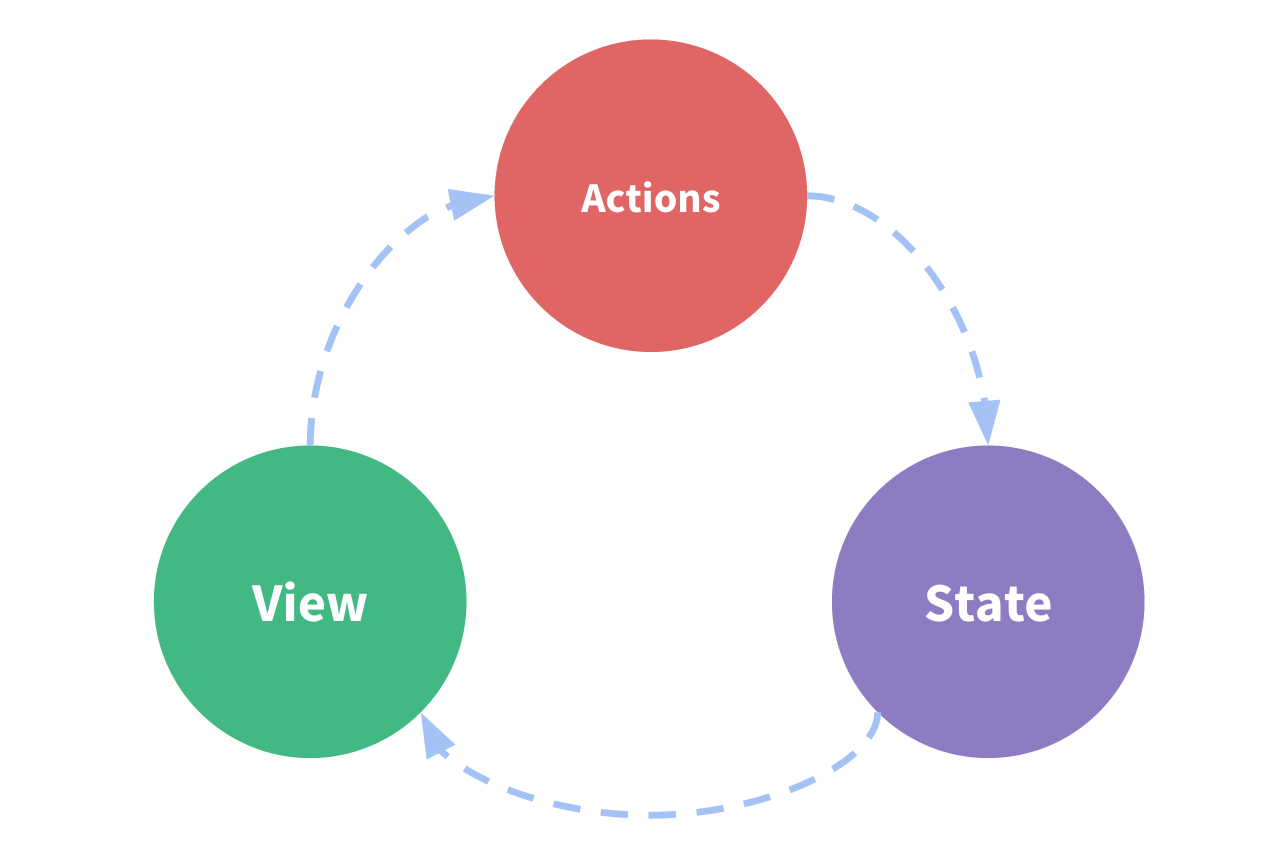
**The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur**. Redux guides you towards writing code that is predictable and testable, which helps give you confidence that your application will work as expected.

### When Should I Use Redux?[​](https://redux.js.org/tutorials/essentials/part-1-overview-concepts#when-should-i-use-redux)

Redux helps you deal with shared state management, but like any tool, it has tradeoffs. There are more concepts to learn, and more code to write. It also adds some indirection to your code, and asks you to follow certain restrictions. It's a trade-off between short term and long term productivity.

Redux is more useful when:

* You have large amounts of application state that are needed in many places in the app
* The app state is updated frequently over time
* The logic to update that state may be complex
* The app has a medium or large-sized codebase, and might be worked on by many people



#### Actions[​](https://redux.js.org/tutorials/essentials/part-1-overview-concepts#actions)

An **action** is a plain JavaScript object that has a type field. **You can think of an action as an event that describes something that happened in the application**.

A typical action object might look like this:

const addTodoAction = {  
 type: 'todos/todoAdded',  
 payload: 'Buy milk'  
}

#### Action Creators[​](https://redux.js.org/tutorials/essentials/part-1-overview-concepts#action-creators)

An **action creator** is a function that creates and returns an action object. We typically use these so we don't have to write the action object by hand every time:

const addTodo = text => {  
 return {  
 type: 'todos/todoAdded',  
 payload: text  
 }  
}

#### Reducers[​](https://redux.js.org/tutorials/essentials/part-1-overview-concepts#reducers)

A **reducer** is a function that receives the current state and an action object, decides how to update the state if necessary, and returns the new state: (state, action) => newState. **You can think of a reducer as an event listener which handles events based on the received action (event) type.**

Reducers must *always* follow some specific rules:

* They should only calculate the new state value based on the state and action arguments
* They are not allowed to modify the existing state. Instead, they must make *immutable updates*, by copying the existing state and making changes to the copied values.
* They must not do any asynchronous logic, calculate random values, or cause other "side effects"

const initialState = { value: 0 }  
  
function counterReducer(state = initialState, action) {  
 // Check to see if the reducer cares about this action  
 if (action.type === 'counter/increment') {  
 // If so, make a copy of `state`  
 return {  
 ...state,  
 // and update the copy with the new value  
 value: state.value + 1  
 }  
 }  
 // otherwise return the existing state unchanged  
 return state  
}

#### Dispatch[​](https://redux.js.org/tutorials/essentials/part-1-overview-concepts#dispatch)

The Redux store has a method called dispatch. **The only way to update the state is to call store.dispatch() and pass in an action object**. The store will run its reducer function and save the new state value inside, and we can call getState() to retrieve the updated value:

store.dispatch({ type: 'counter/increment' })

# connect()

## Overview[​](https://react-redux.js.org/api/connect#overview)

The connect() function connects a React component to a Redux store.

It provides its connected component with the pieces of the data it needs from the store, and the functions it can use to dispatch actions to the store.

function connect(mapStateToProps?, mapDispatchToProps?)

The mapStateToProps and mapDispatchToProps deals with your Redux store’s state and dispatch, respectively. state and dispatch will be supplied to your mapStateToProps or mapDispatchToProps functions as the first argument.

The returns of mapStateToProps and mapDispatchToProps are referred to internally as stateProps and dispatchProps, respectively.

// do not pass `mapDispatchToProps`  
connect()(MyComponent)  
connect(mapState)(MyComponent)

If your mapDispatchToProps is declared as a function taking one parameter, it will be given the dispatch of your store.

const mapDispatchToProps = (dispatch) => {  
 return {  
 // dispatching plain actions  
 increment: () => dispatch({ type: 'INCREMENT' }),  
 decrement: () => dispatch({ type: 'DECREMENT' }),  
 reset: () => dispatch({ type: 'RESET' }),  
 }  
}

##### ownProps[​](https://react-redux.js.org/api/connect#ownprops-1)

If your mapDispatchToProps function is declared as taking two parameters, it will be called with dispatch as the first parameter and the props passed to the wrapper component as the second parameter, and will be re-invoked whenever the connected component receives new props.

The second parameter is normally referred to as ownProps by convention.

// binds on component re-rendering  
<button onClick={() => this.props.toggleTodo(this.props.todoId)} />  
  
// binds on `props` change  
const mapDispatchToProps = (dispatch, ownProps) => ({  
 toggleTodo: () => dispatch(toggleTodo(ownProps.todoId)),  
})

Using the connect function

function mapStateToProps(state) {  
 return { todos: state.todos }  
}  
  
export default connect(mapStateToProps)(TodoApp)

# Writing Logic with Thunks

### What is a "thunk"?[​](https://redux.js.org/usage/writing-logic-thunks#what-is-a-thunk)

The word "thunk" is a programming term that means ["a piece of code that does some delayed work"](https://en.wikipedia.org/wiki/Thunk). Rather than execute some logic *now*, we can write a function body or code that can be used to perform the work *later*.

For Redux specifically, **"thunks" are a pattern of writing functions with logic inside that can interact with a Redux store's dispatch and getState methods**.

Using thunks requires the [redux-thunk middleware](https://github.com/reduxjs/redux-thunk) to be added to the Redux store as part of its configuration.

A thunk function may contain *any* arbitrary logic, sync or async, and can call dispatch or getState at any time.

In the same way that Redux code normally uses [action creators to generate action objects for dispatching](https://redux.js.org/tutorials/fundamentals/part-7-standard-patterns#action-creators) instead of writing action objects by hand, we normally use *thunk action creators* to generate the thunk functions that are dispatched. A thunk action creator is a function that may have some arguments, and returns a new thunk function. The thunk typically closes over any arguments passed to the action creator, so they can be used in the logic:

Thunk action creators and thunk functions

// fetchTodoById is the "thunk action creator"  
export function fetchTodoById(todoId) {  
 // fetchTodoByIdThunk is the "thunk function"  
 return async function fetchTodoByIdThunk(dispatch, getState) {  
 const response = await client.get(`/fakeApi/todo/${todoId}`)  
 dispatch(todosLoaded(response.todos))  
 }  
}

Writing thunks using arrow functions

export const fetchTodoById = todoId => async dispatch => {  
 const response = await client.get(`/fakeApi/todo/${todoId}`)  
 dispatch(todosLoaded(response.todos))  
}

In either case, the thunk is dispatched by calling the action creator, in the same way as you'd dispatch any other Redux action:

function TodoComponent({ todoId }) {  
 const dispatch = useDispatch()  
  
 const onFetchClicked = () => {  
 // Calls the thunk action creator, and passes the thunk function to dispatch  
 dispatch(fetchTodoById(todoId))  
 }  
}

### Why Use Thunks?

Redux reducers [must not contain side effects](https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers#rules-of-reducers), but real applications require logic that has side effects. Some of that may live inside components, but some may need to live outside the UI layer. Thunks (and other Redux middleware) give us a place to put those side effects.

It's common to have logic directly in components, such as making an async request in a click handler or a useEffect hook and then processing the results. However, it's often necessary to move as much of that logic as possible outside the UI layer. This may be done to improve testability of the logic, to keep the UI layer as thin and "presentational" as possible, or to improve code reuse and sharing.