



REACT JS



# JavaScript Essentials

# Variables

JavaScript variables are containers for storing data values.

In this example, x, y, and z, are variables, declared with the var keyword:

```
var x = 5;
```

```
var y = 6;
```

```
var z = x + y;
```

# Type of variable

1:) Var

2:) Let

3:) Const

## JavaScript - Switch Case

Use the switch statement to select one of many code blocks to be executed.

```
switch(expression) {
```

```
  case x:
```

```
    // code block
```

```
    break;
```

```
  case y:
```

```
    // code block
```

```
    break;
```

```
  default:
```

```
let yourFood = prompt("Enter Your Food :");  
switch (yourFood) {  
  case "hello":  
    console.log("Hello");  
    break;  
  
  case "banana":  
    console.log("Your have order banana");  
    break;  
}
```

# Conditional branching: if, '?'

Sometimes, we need to perform different actions based on different conditions.

To do that, we can use the `if` statement and the conditional operator `?`, that's also called a “question mark” operator.

## The if Statement

Use the `if` statement to specify a block of JavaScript code to be executed if a condition is true.

```
if (condition) {  
    // block of code to be executed if the condition is true  
}
```

## The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is false.

```
if (condition) {  
    // block of code to be executed if the condition is true  
} else {  
    // block of code to be executed if the condition is false  
}
```



## Conditional (ternary) operator

The **conditional (ternary) operator** is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and finally the expression to execute if the condition is falsy. This operator is frequently used as a shortcut for the `if` statement.

`isMember ? '$2.00' : '$10.00'`

# JavaScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value.

# The For Loop

- for - loops through a block of code a number of times

```
for (statement 1; statement 2; statement 3) {  
    // code block to be executed  
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

# Example

```
for (i = 0; i < 5; i++) {  
    text += "The number is " + i + "<br>";  
}
```

# Nested Loops

Nested Loop is a loop that is present inside another loop. Javascript supports the nested loop in javascript. The loop can have one or more or simple can have any number of loops defined inside another loop, and also can behave n level of nesting inside the loop.

```
Outerloop
```

```
{
```

```
Innerloop
```

```
{
```

```
// statements to be execute inside inner loop
```

```
}
```

```
// statements to be execute inside outer loop
```

```
}
```

# Let Do this program

```
for(let j = 1 ; j <= 5 ; j++){  
  for(let i = 1 ; i <= j ; i++){  
    document.write("*")  
  }  
  document.write(`<br>`)  
}
```

# The While Loop

The while loop loops through a block of code as long as a specified condition is true.

```
while (condition) {  
    // code block to be executed  
}
```

```
while (i < 10) {  
    text += "The number is " + i;  
    i++;
```

# Example

```
let rep = 1;  
while(rep <= 10){  
    console.log("hello" + rep)  
    rep++  
}
```



# Another Example

```
let dice = Math.trunc(Math.random() * 6) + 1;

while(dice !== 6){
  console.log("Hello " + dice)
  dice = Math.trunc(Math.random() * 6) + 1;
}
```

# JSON

JSON stands for JavaScript Object Notation

JSON is a text format for storing and transporting data

JSON is "self-describing" and easy to understand

What is JSON?

- JSON stands for JavaScript Object Notation
- JSON is a lightweight data-interchange format
- JSON is plain text written in JavaScript object notation
- JSON is used to send data between computers
- JSON is language independent \*

# JSON Syntax

The JSON syntax is a subset of the JavaScript syntax.

## JSON Syntax Rules

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

## JSON Data - A Name and a Value

**JSON data is written as name/value pairs (aka key/value pairs).**

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

Example

```
"name":"John"
```

### **JSON - Evaluates to JavaScript Objects**

The JSON format is almost identical to JavaScript objects.

In JSON, keys must be strings, written with double quotes:

JSON

```
{"name":"John"}
```

# JSON Data Types

## Valid Data Types

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- null

## JSON Strings

Strings in JSON must be written in double quotes.

```
{"name":"John"}
```

## JSON Numbers

```
{"age":30}
```

## JSON Objects

Values in JSON can be objects.

```
{  
  "employee":{"name":"John", "age":30, "city":"New York"}  
}
```

## JSON Arrays

Values in JSON can be arrays.

```
{  
  "employees": [ "John", "Anna", "Peter" ]  
}
```

# JSON.parse()

A common use of JSON is to exchange data to/from a web server.

When receiving data from a web server, the data is always a string.

Parse the data with JSON.parse(), and the data becomes a JavaScript object.

```
const obj = JSON.parse('{ "name": "John", "age": 30, "city": "New York" }');
```



# JSON.stringify()

```
const obj = {name: "John", age: 30, city: "New York"};
```

Use the JavaScript function `JSON.stringify()` to convert it into a string.

```
const myJSON = JSON.stringify(obj);
```

# Example

```
let myJson = `{
  "name": "javascript",
  "age": 25,
  "product" : [
    {
      "id" : 1,
      "name" : "golul",
      "surName" : "agrawal"
    },
    {
      "id" : 2,
      "name" : "sunny",
      "surName" : "yadav"
    }
  ]
}`;

let jsonFile = JSON.parse(myJson);
```

```
console.log(jsonFile.product);  
let table = document.getElementById("table");  
  
for (const data of jsonFile.product) {  
  const html = `  
    <tr>  
      <td> ${data.id} </td>  
      <td> ${data.name} </td>  
      <td> ${data.surName} </td>  
    </tr>  
  `;  
  
  table.insertAdjacentHTML("beforeend", html);  
}
```

# Functions

Quite often we need to perform a similar action in many places of the script.

For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.

Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

We’ve already seen examples of built-in functions, like `alert(message)`, `prompt(message, default)` and `confirm(question)`. But we can create functions of our own as well.

# Function Declaration

To create a function we can use a function declaration.

It looks like this:

```
function showMessage() {  
    alert( 'Hello everyone!' );  
}
```

our new function can be called by its name: showMessage().

# Local variables

A variable declared inside a function is only visible inside that function.

```
function showMessage() {  
    let message = "Hello, I'm JavaScript!"; // local variable  
  
    alert( message );  
}  
  
showMessage(); // Hello, I'm JavaScript!  
  
alert( message ); // <-- Error! The variable is local to the function
```

# Outer variables

A function can access an outer variable as well, for example:

```
let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John
```

# Parameters

We can pass arbitrary data to functions using parameters (also called function arguments) .

```
function showMessage(from, text) { // arguments: from, text
    alert(from + ': ' + text);
}
```

```
showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```



# Returning a value

A function can return a value back into the calling code as the result.

```
function sum(a, b) {  
    return a + b;  
}  
  
let result = sum(1, 2);  
alert( result ); // 3
```

## A Function as an Expression

A function as an expression can be created as shown in the following code example.

```
let add = function a(num1,num2){  
    let sum = num1+ num2;  
    return sum;  
}
```

```
let res = add(8,9);  
console.log(res);// 17
```

## An Arrow Function

The arrow functions were introduced in ECMA 2015 with the main purpose of giving a shorter syntax to a function expression.

```
var add = (num1, num2) => num1+num2;  
let res = add(5,2);  
console.log(res); // 7
```

# Arrays

Objects allow you to store keyed collections of values. That's fine.

But quite often we find that we need an ordered collection, where we have a 1st, a 2nd, a 3rd element and so on. For example, we need that to store a list of something: users, goods, HTML elements etc.

It is not convenient to use an object here, because it provides no methods to manage the order of elements. We can't insert a new property "between" the existing ones. Objects are just not meant for such use.

There exists a special data structure named Array, to store ordered collections.

There are two syntaxes for creating an empty array:

```
let arr = new Array();  
let arr = [];
```

Almost all the time, the second syntax is used. We can supply initial elements in the brackets:

Array elements are numbered, starting with zero.

We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
alert( fruits[0] ); // Apple
```

```
alert( fruits[1] ); // Orange
```

```
alert( fruits[2] ); // Plum
```

We can replace an element:

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

...Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```



# An array can store elements of any type.

```
// mix of values
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];

// get the object at index 1 and then show its name
alert( arr[1].name ); // John

// get the function at index 3 and run it
arr[3](); // hello
```

Methods pop/push, shift/unshift

## Iterate: forEach

The [arr.forEach](#) method allows to run a function for every element of the array.

```
arr.forEach(function(item, index, array) {  
    // ... do something with item  
});
```

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {  
  alert(`${item} is at index ${index} in ${array}`);  
});
```

# Transform an array

Let's move on to methods that transform and reorder an array.

## **map**

The arr.map method is one of the most useful and often used.

It calls the function for each element of the array and returns the array of results.

```
let result = arr.map(function(item, index, array) {  
    // returns the new value instead of item  
});
```

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);  
alert(lengths); // 5,7,6
```

# filter

The find method looks for a single (first) element that makes the function return true.

If there may be many, we can use [arr.filter\(fn\)](#).

The syntax is similar to find, but filter returns an array of all matching elements:

```
let results = arr.filter(function(item, index, array) {  
  // if true item is pushed to results and the iteration continues  
  // returns empty array if nothing found  
});
```

```
let users = [  
  {id: 1, name: "John"},  
  {id: 2, name: "Pete"},  
  {id: 3, name: "Mary"}  
];  
  
// returns array of the first two users  
let someUsers = users.filter(item => item.id < 3);  
  
alert(someUsers.length); // 2
```



# Spread Operator

## Syntax

```
Const arr = [7,5,9]
```

```
Const newArray = [1,2, ...arr]
```

```
console.log(newArray)
```

# Two Important use case of spread operator

1:) to create shallow copies of arrays

2:) to merge two arrays together

//copy array

```
const rest = {  
  mainMenu : ['pizza' , 'burger' , 'mango']  
}
```

```
const restMenuCopy = [...rest.mainMenu]  
console.log(restMenuCopy)
```

merge

```
const aar1 = [1,2,3,4]
const aar2 = [5,6,7,8]

const array = [...aar1 , ...aar2]

console.log(array)
```

# Class basic syntax

In object-oriented programming, a class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).

In practice, we often need to create many objects of the same kind, like users, or goods or whatever.

But in the modern JavaScript, there's a more advanced “class” construct, that introduces great new features which are useful for object-oriented programming.

In JavaScript, classes are the special type of functions. We can define the class just like function declarations and function expressions.

The JavaScript class contains various class members within a body including methods or constructor.

The class syntax contains two components:

- Class declarations
- Class expressions

# The “class” syntax

```
class MyClass {  
    // class methods  
    constructor() { ... }  
    method1() { ... }  
    method2() { ... }  
    method3() { ... }  
    ...  
}
```

Then use `new MyClass()` to create a new object with all the listed methods.

The `constructor()` method is called automatically by `new`, so we can initialize the object there.

```
class User {  
  
  constructor(name) {  
    this.name = name;  
  }  
  
  sayHi() {  
    alert(this.name);  
  }  
  
}  
  
// Usage:  
let user = new User("John");  
user.sayHi();
```



When new User("John") is called:

1. A new object is created.
2. The constructor runs with the given argument and assigns it to this.name.

Then we can call object methods, such as user.sayHi().

# Class inheritance

Class inheritance is a way for one class to extend another class.

So we can create new functionality on top of the existing.

# The “extends” keyword

Let say we have a class Student

```
class Student {  
  constructor(name , age){  
    this.name = name  
    this.age = age  
  }  
  
  bioData(){  
    console.log(`Hi i am ${this.name}`)  
  }  
}  
  
let obj1 = new Student('parth' , 22)  
  
obj1.bioData()
```

```
class Player extends Student{
  constructor(name, age , game){
    super(name,age)
    this.game = game
  }

  playerBioData(){
    console.log(`my name is ${this.name} and i play ${this.game}`)
  }
}
```

```
let obj2 = new Player('satyam' , 22 , 'Football')
let obj3 = new Player('Parth' , 22 , 'Cricket')
obj2.playerBioData()
obj2.bioData()
obj3.playerBioData()
```

# Promise

Imagine that you're a top singer, and fans ask day and night for your upcoming song.

To get some relief, you promise to send it to them when it's published. You give your fans a list. They can fill in their email addresses, so that when the song becomes available, all subscribed parties instantly receive it. And even if something goes very wrong, say, a fire in the studio, so that you can't publish the song, they will still be notified.

Everyone is happy: you, because the people don't crowd you anymore, and fans, because they won't miss the song.

```
let promise = new Promise(function(resolve, reject) {  
  // executor (the producing code, "singer")  
});
```

The function passed to `new Promise` is called the executor. When `new Promise` is created, the executor runs automatically. It contains the producing code which should eventually produce the result. In terms of the analogy above: the executor is the “singer”.

Its arguments `resolve` and `reject` are callbacks provided by JavaScript itself. Our code is only inside the executor.

When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

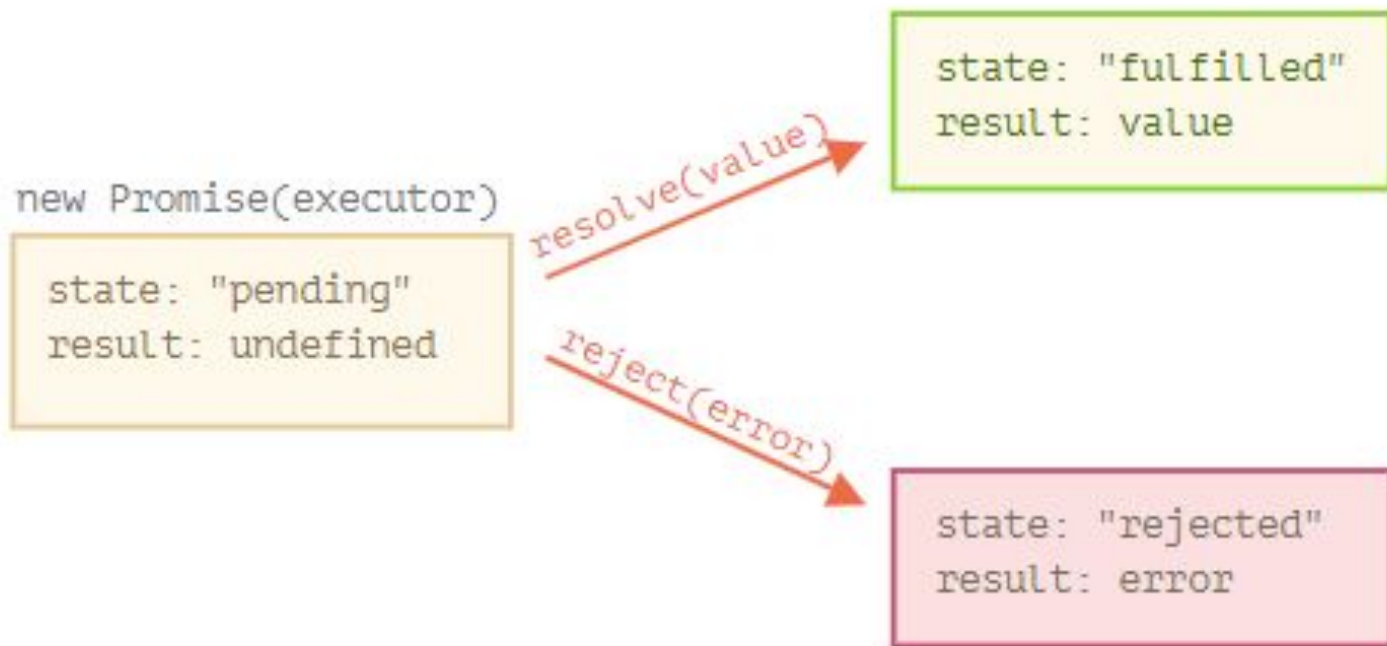
- `resolve(value)` — if the job is finished successfully, with result value.
- `reject(error)` — if an error has occurred, `error` is the error object.

So to summarize: the executor runs automatically and attempts to perform a job. When it is finished with the attempt, it calls `resolve` if it was successful or `reject` if there was an error.

- state — initially "pending", then changes to either "fulfilled" when `resolve` is called or "rejected" when `reject` is called.
- result — initially undefined, then changes to value when `resolve(value)` called or error when `reject(error)` is called.



these states.



# Consumers: then, catch

A Promise object serves as a link between the executor (the “producing code” or “singer”) and the consuming functions (the “fans”), which will receive the result or error. Consuming functions can be registered (subscribed) using methods `.then`, `.catch`

# Then

The most important, fundamental one is `.then`.

# Catch

If we're interested only in errors

```
const greaterThen = new Promise(function (resolve, reject) {  
  setTimeout(function () {  
    if (Math.random() >= 0.5) {  
      resolve("you Win");  
    } else {  
      reject(new Error("you lost"));  
    }  
  }, 2000);  
});
```

```
greaterThen.then(res => console.log(res)).catch(err => console.log(err))
```

# Fetching an API

application programming interface (API)

Piece of Software that can be used by other piece of software in order to allow application to talk to each other

# Example

```
let contryDetails = fetch("https://restcountries.eu/rest/v2/name/eesti")
  .then((res) => res.json())
  .then((data) => {
    console.log(data[0]);
  })
  .catch((err) => console.log(err));
```

# React Js

# What is React?

“React is an open source javascript library for building user interfaces”

Two things here in definition (1) React is a javascript **library**. It is not a framework it is a library (2) It focus on doing **building user Interfaces**.

So react does not focus on the other aspect of your application like routing or http request. It is responsible for only for building rich user interfaces.

You don't have to worry about routing and http, React has rich ecosystem place very well with other libraries and more capable of building full web applications



# Why learn React?

React is a project create and maintain by Facebook.

When a company like facebook uses react in its own product and invest money and resources to keep project alive you can be rest of sure it will be not going to die down anytime soon.

You will found lots of articles & stack overflow solution for most of the problems.

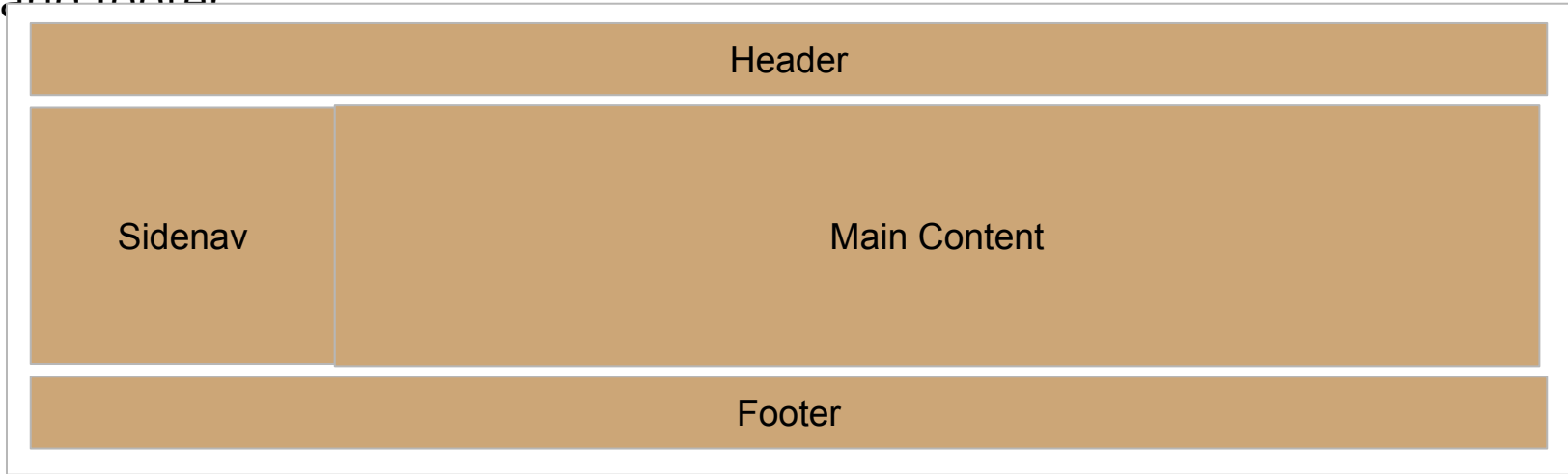
React has been incredibly popular among developers and it is one of the most sort of skill set by company right now.

Learn react and you have a great chance of landing that front end developer trip you always wanted.

# Component Based Architecture

This lets you break down application into small encapsulated parts which can then be composed to make more complex UI.

Ex: traditional website can be broken down into header, side nav, main content and footer



Each section represent a component

# Getting ready

First, we need to create our React application using `create-react-app`. Once that is done, you can proceed to create your first React component.

Before you install `create-react-app`, remember that you need to download and `install Node` from [www.nodejs.org](https://www.nodejs.org). You can install it for Mac, Linux, and Windows.

## **Install react**

```
npx create-react-app my-app
```

- Go to the new application with `cd my-first-react-app` and start it with `npm start`.
- The application should now be running at <http://localhost:3000>.
- Create a new file called `Home.js` inside your `src` folder:

```
import React, { Component } from 'react';

class Home extends Component {
  render() {
    return <h1>I'm Home Component</h1>;
  }
}

export default Home;
```

- You may have noticed that we are exporting our class component at the end of the file, but it's fine to export it directly on the class declaration, like this:

```
import React, { Component } from 'react';  
export default class Home extends Component {  
  render() {  
    return <h1>I'm Home Component</h1>;  
  }  
}
```

- Now that we have created the first component, we need to render it. So we need to open the App.js file, import the Home component, and then add it to the render method of the App component
- Let's change this code a little bit. As I said before, we need to import our Home component and then add it to the JSX. We also need to replace the
- element with our component, like this:

```
import React, { Component } from 'react';
import logo from './logo.svg';

// We import our Home component here...
import Home from './Home';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        { /* Here we add our Home component to be render it */ }
        <Home />
      </div>
    );
  }
}

export default App;
```

```
import ReactDOM from 'react-dom'  
import App from './App'  
  
ReactDOM.render(<App /> , document.getElementById('root'))
```

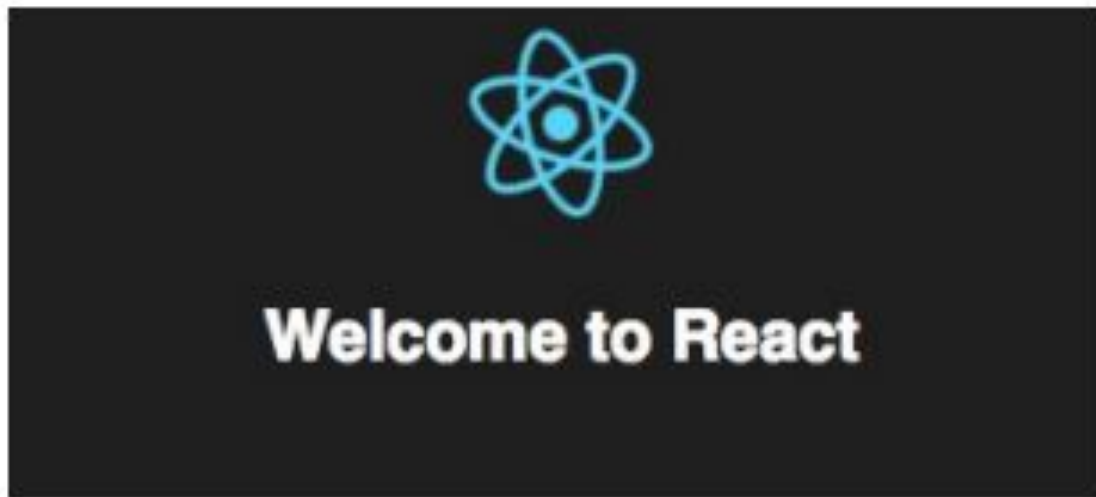


# How it works...

As you can see, we imported `React` and `Component` from the `React` library. You probably noticed that we are not using the `React` object directly. To write code in `JSX`, you need to import `React`. `JSX` is similar to `HTML`, but with a few differences.

This component is called a class component (`React.Component`), and there are functional components, also known as stateless components, which we will cover in the following

If you run the application, you should see something like this



**I'm Home Component**

In our example, we created the Home.js file, and our component's name is Home.

*All React component names should start with the first letter capitalized in both the file and the class name. To begin with, it might feel uncomfortable for you to see this, but this is the best practice in React.*

Some of the main differences between JSX and HTML are the attributes names. You may have noticed that we are using className instead of class. This is the only special attribute name. Others that are two words separated by a dash need to be converted to camelCase, for example, onClick, srcSet, and tabIndex.

# Styling a component with CSS classes and inline styles

Now let's add some CSS to our Home component.

**How to do it...**

- Create a new application, or use the previous one (my-first-react-app).
- Then create a new CSS file for our Home component.
- Now you need to create a Home.css file at the same level as your Home.js file
- We'll now add styles to our Home.css. Basically, we wrapped our component into a div with a className
- We need to import our Home.css file directly,
- Now let's suppose you need to add an inline style. We do this with the style property, and the CSS properties need to be written in camelCase and between {{ }}

```
<button  
  style={{  
    backgroundColor: 'gray',  
    border: '1px solid black'  
  }}  
>
```

- You also can pass an object to the style property like this

```
render() {  
  // Style object...  
  const buttonStyle = {  
    backgroundColor: 'gray',  
    border: '1px solid black'  
  };  
}
```

```
<p>  
  <button style={buttonStyle}>Click me!</button>  
</p>
```

# local state in a component

The local state is a fundamental feature of React for creating dynamic components. Local state is only available on class components, and each component manages its state. You can define the initial value of the state on the component's constructor, and when you update the value of the state, the component will be re-render itself.

Local state is helpful with toggles, for handling forms, and is used to manage information within the same component. It is not recommended to use local state if we need to share data between different components.

Let's define our initial state. Let's see how it works the component's render method when the local state is updated:



```
import React, { Component } from 'react';
import './Home.css';

class Home extends Component {
  constructor() {
    // We need to define super() at the beginning of the
    // constructor to have access to 'this'
    super();

    // Here we initialize our local state as an object
    this.state = {
      name: 'Carlos'
    };
  }

  render() {
    return (
      <div className="Home">
        { /* Here we render our state name */ }
        <p>Hi my name is {this.state.name}</p>
      </div>
    );
  }
}

export default Home;
```

- In this example, we are defining our local state in the constructor as an object, and in the render, we are printing the value directly. We are using `super()` at the beginning of the constructor. This is used to call the parent constructor, (`React.Component`). If we don't include it, we will get an error
- Updating our local state with `this.setState()`:
- Right now, this is just a state that is not being updated. That means that the component will never rerender again. To update the state, we need to use the `this.setState()` method and pass the new value of the state.

# Understanding React lifecycle methods

React provides methods to handle the data during the lifecycle of a component. This is very useful when we need to update our application at particular times.

# What are React lifecycle methods?

You can think of React lifecycle methods as the series of events that happen from the birth of a React component to its death.

Every component in React goes through a lifecycle of events. I like to think of them as going through a cycle of birth, growth, and death.

- **Mounting** – Birth of your component
- **Update** – Growth of your component
- **Unmount** – Death of your component

Now that we understand the series of lifecycle events let's learn more about how they work.

Life cycle in react means

When any component created and and when it is destroyed

When this component is actually created the constructor going to run

Then render method run

# Common React Lifecycle Methods

## **render()**

The `render()` method is the most used lifecycle method. You will see it in all React classes. This is because `render()` is the only required method within a class component in React.

As the name suggests it handles the rendering of your component to the UI. It happens during the mounting and updating of your component.

Below is an example of a simple `render()` in React.

```
class Hello extends Component{  
  render(){  
    return <div>Hello {this.props.name}</div>  
  }  
}
```

# componentDidMount()

Now your component has been mounted and ready, that's when the next React lifecycle method `componentDidMount()` comes in play.

`componentDidMount()` is called as soon as the component is mounted and ready. This is a good place to initiate API calls, if you need to load data from a remote endpoint.

Unlike the `render()` method, `componentDidMount()` allows the use of `setState()`. Calling the `setState()` here will update state and cause another rendering but it will happen before the browser updates the UI. This is to ensure that the user will not see any UI updates with the double rendering.



# componentDidUpdate()

This lifecycle method is invoked as soon as the updating happens. The most common use case for the `componentDidUpdate()` method is updating the DOM in response to prop or state changes.

# componentWillUnmount()

As the name suggests this lifecycle method is called just before the component is unmounted and destroyed. If there are any cleanup actions that you would need to do, this would be the right spot.

# Function Component

# What are Functional Components?

There are two main types of components in React. Class Components and Functional Components. The difference is pretty obvious. Class components are ES6 classes and Functional Components are functions.

# PROPS

Props stand for "Properties." They are read-only components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

```
import Welcome from './Welcome';

function App() {
  return (
    <div className="App">
      <Welcome name="John"/>
      <Welcome name="Mary"/>
      <Welcome name="Alex"/>
    </div>
  );
}
```

Props are custom values and they also make components more dynamic. Since the Welcome component is the child here, we need to define props on its parent (App), so we can pass the values and get the result simply by accessing the prop "name":

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```



# Hooks

Hooks provide the most commonly needed functionalities in stateful React apps.

They are as follows:

- `useState`
- `useEffect`
- `useContext`

# useState

The useState Hook is used to deal with state in React. We can use it as follows:

```
import { useState } from 'react'
```

```
const [ state, setState ] = useState(initialState)
```

The useState Hook replaces this.state and this.setState()

# useEffect

This Hook works similarly to adding a function on `componentDidMount` and `componentDidUpdate`. Furthermore, the Effect Hook allows for returning a cleanup function from it, which works similarly to adding a function to `componentWillUnmount`

The `useEffect` Hook is used to deal with effectful code, such as timers, subscriptions, requests, and so on. We can use it as follows:

```
import { useEffect } from 'react'

useEffect (didUpdate)
```

The `useEffect` Hook replaces the `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` methods

# React Hooks – What is it trying to solve?

Hooks were introduced in React version 16.8 and now used by many teams that use React.

Hooks solves the problem of code reuse across components. They are written without classes. This does not mean that React is getting rid of classes, but hooks is just an alternate approach.

# How to use State hook?

now let's dive deep into the useState hook

The purpose of the useState hook is to add state to your functional components in React, without having to use a class.

# Traditional Class Component

Let's look into an example to see how we can convert a class component into a functional component and manage state using the useState hook.

I am creating a YesNoComponent, that contains two buttons (Yes and No). When Yes is pressed, it sets the buttonPressed state to "Yes", and when No is pressed, it sets the buttonPressed state to "No".



```
class YesNoComponent extends React.Component {
  state = {
    buttonPressed: ""
  };

  onYesPress() {
    this.setState({ buttonPressed: "Yes" });
    console.log("Yes was pressed");
  }

  onNoPress() {
    this.setState({ buttonPressed: "No" });
    console.log("No was pressed");
  }

  render() {
    return (
      <div>
        <button onClick={() => this.onYesPress()}>Yes</button>
        <button onClick={() => this.onNoPress()}>No</button>
      </div>
    );
  }
}
```

# Functional Component with Hooks

Now let's rewrite this component into a functional component. We are going to get rid of the class and instead just use a functional component. You may ask, then how do we manage the local state? Well, that's where we are going to use React's `useState` hook.

```
import React, { useState } from "react";

const YesNoComponentFunctional = () => {
  const [button, setButton] = useState("");

  const onYesPress = () => {
    setButton("Yes");
    console.log({ button });
  };

  const onNoPress = () => {
    setButton("No");
    console.log({ button });
  };

  return (
    <div>
      <button onClick={() => onYesPress()}>Yes</button>
      <button onClick={() => onNoPress()}>No</button>
    </div>
  );
};

export default YesNoComponentFunctional;
```

# Difference

The above component generates the same result as the previous YesNoComponent. The only difference is that this is a functional component while the other was written in a class.

# useState Details

There are few important pieces in this code that made this happen. Let's look at each one of them.

- The first thing we need to do is **import** the useState from React.

```
import React, { useState } from "react";
```

- The next step is to **declare** the state variable for the component

```
const [button, setButton] = useState("");
```

Here we have declared a new state variable called button. The useState then sets the initial value for button here as an empty string "". useState is the same as this.state in our class components.

# What does useState return?

It returns a pair of values: the current state and a function that updates it. In our example, the current state is `button` and the function that updates it is `setButton`. Here `setButton` is going to behave the same as `setState` from the previous example.

Notice in the `onYesPress()` and `onNoPress()` functions the `setButton` function is used to set the current state of the button.

# How to use Effect hook?

useEffect hook essentially is to allow side effects within the functional component.

In class components, you may be familiar with **lifecycle methods**. The lifecycle methods, `componentDidMount`, `componentDidUpdate` and `componentWillUnmount`, are all handled by the `useEffect` hook in functional components.

Before the introduction of this hook, there was no way to perform these side-effects in a functional component. Now the `useEffect` hook, can provide the same functionality as the three lifecycle methods mentioned above. Let's look at some examples to learn this better.



# Before Hooks

```
import React from "react";

class TraditionalComponent extends React.Component {
  state = {
    buttonPressed: "",
  };

  componentDidMount() {
    console.log("Component did mount", this.state.buttonPressed)
  }

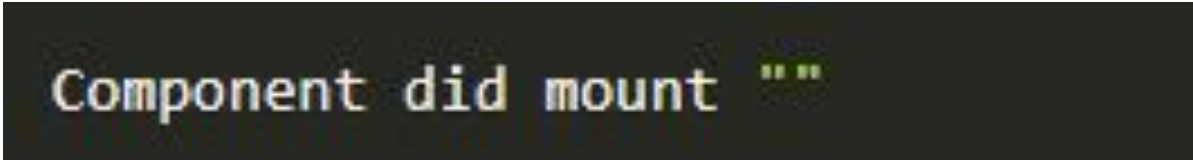
  componentDidUpdate() {
    console.log("Component did update", this.state.buttonPressed)
  }

  onYesPress() {
    this.setState({ buttonPressed: "Yes" });
  }
}
```

```
  onNoPress() {
    this.setState({ buttonPressed: "No" });
  }

  render() {
    return (
      <div>
        <button onClick={() => this.onYesPress()}>Yes</button>
        <button onClick={() => this.onNoPress()}>No</button>
      </div>
    );
  }
}
```

In the example above we have coded a traditional class React component. In class components, we have access to the lifecycle methods. Here I am using `componentDidMount()` and `componentDidUpdate()` with console logs in each of them. When you run this above code, and look at the console you will initially see the following message:



```
Component did mount ""
```

`componentDidMount()` is called as soon as the component is mounted and ready. This is a good place to initiate API calls, if you need to load data from a remote endpoint.

Now if we press the Yes button or the No button, the button state is updated. At this point you should see the following on the console:

```
Component did update Yes  
Component did update Yes
```

The `componentDidUpdate()` method is called when the state changes. This lifecycle method is invoked as soon as the updating happens. The most common use case for the `componentDidUpdate()` method is updating the DOM in response to state changes.

Alright, now what does the `useEffect` hook really do?

# Functional Component with useEffect Hook

Let's now re-write our example into a functional component.

```
import React, { useState, useEffect } from "react";

const UseEffectExample = () => {
  const [button, setButton] = useState("");

  //useEffect hook
  useEffect(() => {
    console.log("useEffect has been called!", button);
  });

  const onYesPress = () => {
    setButton("Yes");
  };

  const onNoPress = () => {
    setButton("No");
  };
}
```

```
return (
  <div>
    <button onClick={() => this.onYesPress()}>Yes</button>
    <button onClick={() => this.onNoPress()}>No</button>
  </div>
);

export default UseEffectExample;
```

We have now rewritten our class component into a functional component.

Note: The first thing we need to do to get the useEffect to work is, **import** the useEffect from React.

```
import React, { useEffect } from "react";
```



Notice here that the `useEffect` hook has access to the state. When you run this code, you will initially see that the `useEffect` is called which could be similar to the `componentDidMount`. After that every time the state of the button changes, the `useEffect` hook is called. This is similar to the `componentDidUpdate` lifecycle.

```
// Console  
useEffect has been called! "" // comparable to componentDidMount  
useEffect has been called! Yes // comparable to componentDidUpdate  
useEffect has been called! No // comparable to componentDidUpdate
```

# Passing Empty Array to useEffect Hook

You can optionally pass an empty array to the useEffect hook, which will tell React to run the effect only when the component mounts.

Here is the modified useEffect hook from the previous example, which will occur at mount time.

```
//useEffect hook
useEffect(() => {
  console.log("useEffect has been called!", button);
}, []);
```

# UseEffect as UnMount

As i have govern an [] empty array as a argument so it will work as a useEffect mean as a component didMount

In a same useEffect if you give a return as a function then this is working as a component unMounted

# Multiple Return

## Conditional rendering

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.

```
function Conditional(){  
  const [loading , setLoading] = useState(true)  
  
  if(loading){  
    return <h1>Loading</h1>  
  }  
  
  return <h2>Hello There</h2>  
}
```

# Handling Form

The way we handle the React form input is a bit different compared to that of regular HTML.

In HTML, the form inputs keep their internal state (i.e the input value) and are maintained by the DOM. But in React, you will be in charge of handling these inputs.

# We know how to handle Form in JS

```
Const input = document.getElementById('myText')
```

```
Const inputValue = input.value
```

# Handling in React

- Value
- Onchange



# Value and onChange

To get the value from the user we use `useState` and we have to add `value` attribute to a `input` tag

`onChange` is a event listener will fire every time when we change the input value

# How to handle multiple inputs in React

## The Problem

- look at the starter code below
- This Form component has 5 input fields in total; 5 different states and 5 different onChange inline functions
- This is not exactly DRY code 🙀

## The Solution: Refactoring

Step 1: Add input default values and initialize state

- First, let's add default values to ALL input fields
- How do we do that? We create an object literal with those values and set to empty string
- Then, with the `useState()` React Hook we initialize our values state with the `initialValues` object
- Important: Remember to add the `value` attribute to every input field with its corresponding value (e.g. `values={values.company}`)

```
const initialValues = {
  company: "",
  position: "",
  link: "",
  date: "",
  note: "",
};

export default function Form() {
  const [values, setValues] = useState(initialValues);

  return (
    <form>
      <input
        value={values.company}
        onChange={(e) => setCompany(e.target.value)}
        label="Company"
      />
    </form>
  );
}
```

## Step 2: Handle multiple input change

- The goal here is to handle ALL inputs with a single onChange handler
- In order to update and keep track of our input fields every time they change, we need to create a handleInputChange function (see below)

- What's happening here? (quick recap)
  - a. First, we're using object destructuring to get or extract the name and the value attributes from our inputs (look at the the comments below - they're equivalent)
  - b. Then, we're updating our values state object with the existing values by using the `setValues()` function and the [spread operator](#)
  - c. And finally, we're updating the value of the event that was triggered by that `onChange` with this ES6 syntax: `[name]: value`
  - d. This is a very important step! We need to add a name attribute to our inputs and `[name]: value` here means that we're setting a dynamic name property key (taken from our inputs - e.g. `company: e.target.value`) which will be equal to the value of our current input state.

```
const handleInputChange = (e) => {  
  //const name = e.target.name  
  //const value = e.target.value  
  const { name, value } = e.target;  
  
  setValues({  
    ...values,  
    [name]: value,  
  });  
};  
  
return (  
  <form>  
    <input  
      value={values.company}  
      onChange={handleInputChange}  
      name="company" //IMPORTANT  
      label="Company"  
    />  
  // ...  
)
```

### Step 3: Add handleChange to input fields

- Add the handleChange function to the onChange prop of every input field
- Look at the final code; this is a much cleaner and manageable form

```
const handleChange = (e) => {  
  const { name, value } = e.target;  
  setValues({  
    ...values,  
    [name]: value,  
  });  
};
```



# Using Refs

When it's time to build a form component in React, there are several patterns available to you. One of these patterns involves accessing the DOM node directly using a React feature called refs. In React, a ref is an object that stores values for the lifetime of a component. There are several use cases that involve using refs.

how we can access a DOM node directly with a ref.

import React, { useRef } from "react";

```
export default function AddColorForm({ onNewColor = f => f }) {  
  const txtTitle = useRef();  
  const hexColor = useRef();  
  
  const submit = e => { ... }  
  
  return (...)  
}
```

```
<form onSubmit={submit}>  
  <input ref={txtTitle} type="text" placeholder="color title..."  
required />
```

```
  <input ref={hexColor} type="color" required />  
  <button>ADD</button>  
</form>  
);  
}
```

Here, we set the value for the txtTitle and hexColor refs by adding the ref attribute to these input elements in JSX. This creates a current field on our ref object that references the DOM element directly. This provides us access to the DOM element, which means we can capture its value. When the user submits this form by clicking the ADD button, we'll invoke the submit function:

```
const submit = e => {  
  e.preventDefault();  
  const title = txtTitle.current.value;  
  const color = hexColor.current.value;  
  onNewColor(title, color);  
  txtTitle.current.value = "";  
  hexColor.current.value = "";  
};
```

When we submit HTML forms, by default, they send a POST request to the current URL with the values of the form elements stored in the body. We don't want to do that. This is why the first line of code in the submit function is `e.preventDefault()`, which prevents the browser from trying to submit the form with a POST request. Next, we capture the current values for each of our form elements using their refs. These values are then passed up to this component's parent via the `onNewColor` function property. Both the title and the hexadecimal value for the new color are passed as function arguments. Finally, we reset the value attribute for both inputs to clear the data and prepare the form to collect another color.

# useContext

This Hook accepts a context object and returns the current context value.

The useContext Hook is used to deal with context in React.

We can use it as follows:

```
import { useContext } from 'react'
```

```
const value = useContext(MyContext)
```

The useContext Hook replaces context consumers.

# What is Context in React?

In React everything is a component, and props are passed from the parent to the child component. This is the general principle of React's component based single directional flow of data approach. But as your application grows, this may become cumbersome and not be ideal for certain global properties that a large number of components may require. For example, if you have user information that you want to display across multiple components, passing props through several components may not be ideal.



This is solved either by using external libraries like Redux which takes the entire state of the application and stores it in the Redux store or we can use React Context. Context provides a way to share values like these between components without having to explicitly pass props through the entire component tree.

# What is useContext Hook?

The React Context API was introduced to overcome the problem of passing props down the tree of components. The state of the application can be stored globally, and shared across multiple components. Now with the addition of hooks to the React architecture, we get a new hook called `useContext`.

The `useContext` hook, just like all the other hooks we have seen before, is easier to write and works with React's functional components.

Alright, let's look into an example to understand this better.

First step is to import the useContext hook.

```
import React, { useContext } from "react";
```

## 1:) Create a context

```
export const userContext = React.createContext()
```

## 2:) Provide a context value

```
<userContext.Provider value={"Siddharth"}>
```

```
  <App/>
```

```
</userContext.Provider>
```

### 3:) Consume a context value with Context hook

```
Import {userContext} from './App'
```

```
Function componentC(){
```

```
    Const user = useContext(userContext)
```

```
    Return (
```

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

```
    )
```

```
}
```

# useReducer

This Hook is an alternative to useState, and works similarly to the Redux library. We can use it as follows:

```
import { useReducer } from 'react'
```

```
const [ state, dispatch ] = useReducer(reducer, initialArg, init)
```

The useReducer Hook is used to deal with complex state logic

# What is a Reducer?

Before, we learn this hook, there is also one more piece that we need to have an understanding on. What is a Reducer ? And how do I write a reducer without messing up?

I like to think of a Reducer in Redux as a “Coffee Maker”. It takes in an old state and action and brews a new state (Fresh coffee).

I like to think of a reducer like a “coffee maker”. The coffee maker takes in coffee powder and water. It then returns a freshly brewed cup of coffee that we can enjoy. Based on this analogy reducers are functions that take in the current state (coffee powder) and actions (water) and brew a new state (fresh coffee).

Reducers are pure functions that take in a state and action and return a new state.

A reducer should always follow the following rules:



# What is the useReducer Hook?

The useReducer is a hook I use sometimes to manage the state of the application. It is very similar to the useState hook, just more complex. It acts as an alternate hook to the useState hook to manage complex state in your application.

The useReducer hook uses the same concept as the reducers in Redux. It is basically a pure function, with no side-effects.

The syntax for the useReducer hook is as follows:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

The useReducer will basically allow React functional components to access the reducer functions from your application's state management.

useReducer have two parameter one is pure function that us reducer and second in initialState

So reducer is a function so let define

```
Const reducer = () => {}
```

And Reducer function passes two argument

```
Const reducer = (state , action) => {  
    return state  
}
```

State is a current value

Action what action we have to provide

Now we can add useReducer hook to a destructuring array

```
Const [state , dispatch] = useReducer(reducer , initialState)
```

So here the state is a initialValue that we have define

Dispatch this is the only way to trigger a state change if the value match

```
<Button onClick={() => dispatch({type : "Increment"})}> + </Button>
```

```
const reducer = (state, action) => {  
  console.log(state, action);  
  if (action.type === "INCREMENT") {  
    return state + 1;  
  }  
  if (action.type === "DECREMENT") {  
    return state - 1;  
  }  
  return state;  
};
```

```
const initialState = 10;
```

```
const [state, dispatch] = useReducer(reducer, initialState);
```



```
<button
  className="btn btn-primary"
  onClick={() => dispatch({ type: "INCREMENT" })}
>
+|
</button>
<button
  className="btn btn-danger"
  onClick={() => dispatch({ type: "DECREMENT" })}
>
-
</button>
```

# UseMemo

useMemo invokes a function to calculate a memoized value. In computer science in general, memoization is a technique that's used to improve performance.

The way useMemo works is that we pass it a function that's used to calculate and create a memoized value

useMemo will only recalculate that value when one of the dependencies has changed

```
const [incrementOne, setincrementOne] = useState(0);
const [incrementTwo, setincrementTwo] = useState(0);

function handlingOne() {
  setincrementOne(incrementOne + 1);
}

function handlingTwo() {
  setincrementTwo(incrementTwo + 1);
}
```

```
<button onClick={handlingOne}> click by {incrementOne} </
button>{" "}
<span>Button one value is {isEven() ? "even" : "odd"} </
span>
<br />
<button onClick={handlingTwo}>click by {incrementTwo}</
button>
```

```
function isEven() {  
  let i = 0;  
  while (i < 999999999) i++;  
  return incrementOne % 2 === 0;  
}
```

```
const isEven = useMemo(() => {  
  let i = 0;  
  while (i < 9999999999) i++;  
  return incrementOne % 2 === 0;  
}, [incrementOne]);
```

# Styling React Components

CSS stylesheets

Inline Styling

CSS Modules

CSS in JS Libraries (styled components)

# styled components

<https://styled-components.com/>

# Bootstrap with React

React-Bootstrap replaces the Bootstrap JavaScript. Each component has been built from scratch as a true React component, without unneeded dependencies like jQuery

<https://react-bootstrap.github.io/>



# Custom Hooks

I am Creating useFetch Custom hook

Why? Because we have to fetch different different API on Different Different Pages

So Now what i will do is to create a custom hook for that

## useFetch.js

Import {useState , useEffect} from 'react'

Export const useFetch = (url) => {

    Const [loading , setLoading] = useState(true)

    Const [product , setProducts] = useState([])

    Const getProduct = **async** => {

        Const resp = await fetch(url)

        Const prod = await resp.json()

        setProducts(prod)

        setLoading(false)

    }

    useEffect(() => {getProduct()} , [url])

    return {loading , product}

}

## App.js

```
Import {useFetch} from "../useFetch";
```

```
Const url = "....."
```

```
Const App = () => {
```

```
    Const {loading , product} = useFetch(url)
```

```
    Return (
```

```
        <h1> {loading ? "Loading....." : "Data"}</h1>
```

```
    )
```

```
}
```

# What is React Router?

Single page applications (SPAs) with multiple views need to have a mechanism of routing to navigate between those different views without refreshing the whole web page. This can be handled by using a routing library such as [React Router](#).

React Router is a library that lets you delineate your routes on the front-end, so a user sees one component of your choice on localhost:3000/homepage, another on localhost:3000/faq, and others on /links, /contact, etc. You can make a React app already, but clicking around now is the name of the game.

# Creating the first route with React Router

let's take a look at how to create routes using the React Router

Search React-router-dom

To create the first route using React Router library, open src/App.js file and add the following import statement:

```
import {  
  BrowserRouter as Router,  
  Switch,  
  Route,  
  Link  
} from "react-router-dom";
```

```
function Main() {  
  return (  
    <div>  
      <Router>  
        { /* All Routes and nested inside it */ }  
      </Router>  
    </div>  
  )  
}
```

The next component to import from react-router-dom is the `Route` and `Switch`:

First create a two component Home and About Component and import it in a Main Component



```
<Router>
  <Switch>
    <Route exact path="/">
      <Home />
    </Route>
    <Route path="/about">
      <About />
    </Route>
  </Switch>
</Router>
```

# Main Component

---

## Home Component

# Main Component

---

I am About

# Adding a navigation menu

To navigate at a particular route within the React app, or the two currently existing routes in the demo app, let's add a minimal navigation bar with the help of the Link component from react-router-dom.

The concept of navigating between different web pages in HTML is to use an anchor tag:

```
<a href="">Some Link Name</a>
```

Using this approach in a React app is going to lead to refreshing a web page, each time a new view or page itself is rendered. This is not the advantage you are looking for when using a library like React. To avoid the refreshing of the web pages, the react-router-dom library provides the Link component.

Next, inside the App function component, create a nav bar

```
<Router>
  <h1>Main Component</h1>
  <Link to="/">Home Page</Link>
  &nbsp; &nbsp; &nbsp;|
  <Link to="/about">About Page</Link>
  <hr />
```

# React Redux

You have finally decided that you are going to use React for your application and you are all pumped up to start development until that fateful moment when you realize that you need data from a different component & the complexity of passing that data through props is making your code & your life HELL.

## **Here comes Redux to your rescue!!**

In simple terms, Redux acts like a container to store the state of your javascript application which is accessible across all your components.

Before we see how we can implement react-redux for an application, we need to understand few terminologies

- 1) The Store
- 2) Action Creators
- 3) Reducers



## The Store

Store is a centralised place where all your application state sits. Consider it as your bank which holds all your cash.

## Action creators

Let's take our previous analogy of Redux Store acting as a bank, now you would definitely want to either withdraw or deposit money right? And what would you do? You would raise a request to do so. Action creators acts as those requests. They emit actions which encapsulates the changes you want to make in the Store.

## Reducers

Now once you have raised a request to deposit or withdraw something, you have to then submit that request to your bank manager

The bank manager knows exactly how to process that request, and if everything looks good, he/she will deposit or withdraw the amount requested, hand it over to you and update the details in your account to reflect the new changes.

This is what Reducers do for you in Redux. They take whatever action you pass, make a copy of the existing state, apply all your updates to the state & finally publish the updated state changes to the store.

```
Import {createStore} from "redux"
```

```
// reducer
```

```
Function reducer(){
```

```
    console.log("Hello Redux")
```

```
}
```

```
Const store = createStore(reducer)
```

Reducer function that is used to update store has two argument state , action

State - old state(state before update)

Action - what happened (what update)

Return updated or old state

So what we do

// initialstore

Const initialStore = {

Count : 0

}

// reducer

Function reducer(state , action){

console.log(state , action)

return state

}

Const store = createStore(reducer , initialStore)

## **getState**

getState we get your state so

When you console.log(store.getState()) you will see all the state

So now i need to pass this state in some component

```
<Header cart={store.getState()} />
```

In Header Component

```
Const Header = ({cart}) => {  
  Const {count} = cart  
  Return (  
    <h1>Cart : {count}</h1>  
  )  
}
```

## First Action

Dispatch method - send action to the store

Action (objects) - must have type property -> what kind of action

Function reducer(state , action

```
{  
    if(action.type === "DECREASE"){  
        return {...state , count : state.count - 1}  
    }  
}
```

```
store.dispatch({type : "DECREASE"})
```

## Action as Variable

We can also pass dispatch type as an variable

```
Const DECREASE = "DECREASE"
```

```
Function reducer(){
```

```
    if(action.type === DECREASE){
```

```
        //code
```

```
    }
```

```
}
```



**We can also placed this variable in another file**

In action.js

Export const DECREASE = "DECREASE"

And import in the file where you want

## **Separate Reducer**

We will keep reducer function in another file and export it

## Provider setup

Let's connect redux to our store

Import {provider} from "react-redux"

And in main div wrap with provider

Const store = createStore(reducer , initialState)

```
<Provider store={store}>
```

```
  <Header />
```

```
</Provider>
```

## Connect Function

In Header component

Import {connect} from "react-redux"

And in export default we have to connect it

```
Const mapStateToProps = state => {
```

```
    return {amount : state.amount}
```

```
}
```

```
Export default connect(mapStateToProps)(Header)
```

## Connect function for dispatch

```
import React from "react";
import { connect } from "react-redux";

function Incdec({ dec , inc , reset }) {
  return (
    <div>
      <button onClick={() => inc()}>+</button>
      <button onClick={() => dec()}>-</button>
      <button onClick={() => reset()}>reset</button>
    </div>
  );
}

const mapDispatchToProps = (dispatch) => {
  return {
    dec: () => dispatch({ type: "DEC" }),
    inc: () => dispatch({ type: "INC" }),
    reset: () => dispatch({ type: "RESET" }),
  };
};
```