Loop through array

(for val of array) //for loop will go through every element of the array with val pointing to it.

Pass all array elements as a function parameter

math.max(...array) pass array values as a parameters (array[0],array[1]...).

Trim function

Trim white spaces at the beg&end of a function

Inline style

Wants an object as an input (between {}) can accept code such as fast if(separate different props using ‘,’)

(javascript values of css)

Template literals ``

Return a string where you can add values to it based on a condition

{‘form ${!istrue ?‘nottrue’ : ‘ ’}`} (if not true then returns form nottrue) \*two classes

$ to insert code between { }

Css modules

Make each class imported from a file.module.css unique to that file only.

(takes each of css classes as a prop)

\*classes with ‘-‘ or ‘\_\_’ can be accessed through className={styles[‘class-name’]}

useState update function

It gives an argument of previous state snapshot in the call.

List <li> id

Each list item should have a key attribute which value is unique.

Pass a value to an event function

Value can’t be passed directly (onClick={probs.onClick(id)}) because it will be summoned with each render .

So instead: onClick={()=>{probs.onClick(user.id)}}

useEffect(callbackfunc,[dependency])

1) useEffect hook without mentioning any dependency array like.. useEffect(someCallbackFuction); runs for every render of the functional component in which its included..  
2) useEffect hook with an empty dependency array like this..  useEffect(callbackFunc , [] ) is executed only for the the initial render of the the functional component. And then it will not run in the further renders of the same functional Component..  
3) useEffect hook with some dependencies inside the dependency array like this.. useEffect(callbackFunc , [dependency] ); will run for the initial render as well as when the render happen due to change in dependencies mentioned in the dependency array...

Replacing div by wrapper (fragments , empty tags)

using muitiple div as wrappers can cause nested div problem so instead use empty tags <> </> or <fragment></fragment>

to use fragment make sure to:

import react,{fragment} from react;

\*fragment can accept attr.

Debounce & Throttle

Debounce: Group multiple repetitive sequential calls in a one call to prevent traffic.

Throttle: allow function to only execute once every X milliseconds.

useReducer()

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

dispatch function take an object with type or more arguments and calls reducer dispatch({ type: 'incremented\_age' });.

function reducer(state, action) {

// ...

}

State: initial state of Arg and action is the object passed within dispatch.

Used when updating more than one related states instead of useState().

setTimeout(func,time(ms))

Delay the execution of a function for X ms.

Can be set to an identifier used in clearTimeout() which will reset the time of function to 0 (restarts the countdown).

useEffect() return value

return a function that executes before the second call (after mounting (component initilization)) and so on.

Also runs during dismount (component destroyed).

\

Context

A way to exchange data between different components without having to pass it down using props.

1. Create context file and a context object which contains a component
2. import React from "react";
3. const AuthContext=React.createContext({
4. isLoggedIn:false // not obligated as you can add more props in provider2
5. });
6. export default AuthContext;

2- provide within the wrapping app using <Authcontext.provider> comp.

<AuthContext.Provider value={{

        isLoggedIn:isLoggedIn

      }}> // must provide a default value within the wrapper

3- consume it within comp. where data is needed

 <AuthContext.Consumer>

       {(ctx)=>{ //ctx is the object containing isLoggedIn

        return( … //code)</AuthContext.Consumer>

// or use  const ctx=useContext(AuthContext);

Const object= useRef(initial)

useRef returns a ref object with a single current property initially set to the initial value you provided.

* You can **store information** between re-renders (unlike regular variables, which reset on every render).
* Changing it **does not trigger a re-render** (unlike state variables, which trigger a re-render).
* The **information is local** to each copy of your component (unlike the variables outside, which are shared).

const intervalId = intervalRef.current; clearInterval(intervalId);

Rendering with state,context and props changes

When a state or a context or props changes the component containing it will be rerendered.

When a component rerenders all the child components revaluates to chick for differences (only revaluated in virtual DOM, real DOM is not affected unless differences are found and only components containing these diff. are rerendered)

memo()

Prevents the revaluating of child components when none of it’s props has been changed.

(props that includes non-permitive js variables are always considered to be changed during paren reredndering because as every variable is recreated again, non-permitive new value doesn’t equal it’s old values.)

Non-permitive data include arrays,objects...

The solution is either to use a useCallack hook that always sets the value of revaluated object to the previous one (hence they’re equal now).

Or include a second argument within memo function a function that always return true.

export default React.memo(Button,()=>true);

State reintialization

A state is never reinitialized (set to initiate value between ()) unless it’s component was completely removed.

Consecutive state updates are batched and handled together.

useMemo(()=>{returns object data},[dep])

Store initial value of object(such as array) in cache & setting each revaluated instance of this object to this stored one such that they’re always equal.

Now when performing time complex operation such as sorting you can use useMemo to preventing performing it each revaluating

 const sortedList = useMemo(() => {

    console.log('Items sorted');

    return items.sort((a, b) => a - b);

  }, [items]); //won’t be performed unless NewITems instance != stored instance

  <DemoList title={listTitleitems={listItems}/>

 const listItems = useMemo(() => [5, 3, 1, 10, 9], []); //make sure to always use the same instance stored in memory so they are always equal hence no reporfoming the sort

Throw {} & catch (error) {}

Surround code which can generate possible error with throw, and then catch the error using catch.

If this code doesn’t throw any kind of error you can create default error with a message property using throw new Error(‘message’); which then will be catched by catch.

then((arg)=>{})

Works with promises when the results of a method isn’t immediate. Take the returning value of a promise function as a default argument.

Async & await

Declaring function a an async function so you can use await instead of the then() chains (more rea

Fetch()

Send an http request to a backend server to fetch data which returns the fetched data.

This is an asynchronous operation means it either needs the chains of then() or asynchronous and await.

If data is in json format use .json() to transform it to plain js which is also async.

useEffect return value

**useEffect offers the use of return function, which is used for cleanup function purposes**

React also cleans up effects from the previous render before running the effects next time.

e.g: in case of racing where u click in list of products, and when each product is clicked you will update the page with this product info, if u click quickly in the products for many times , the useEffect will be updated whenever product\_id is changed, and at some point clicking on product1 [will call api for product1 ]then quickly product2 [will call api for product2 and retrieve data causing -> setProduct(product2data) ] while still data for product1 has just arrived causing -> setProduct(product1data) . This way we have product2 clicked ! but product1 data appears at page!!.

bind

"preconfigure" which arguments that function should receive when it's eventually getting called.

someButton.addEventListener('click', function() {

greet('Max'); // yields 'Hi Max'

});

someButton.addEventListener('click', greet.bind(null, 'Max'));

not all parameters need to be determined as it can be determined during the actuall calling of the function.

For Of

used to iterate over iterable objects, like arrays, strings, maps, sets, etc. It returns the values of the iterable objects, not the keys. It cannot be used with regular objects because they are not iterable.

let arr = ['apple', 'banana', 'cherry'];

for (let value of arr) {

console.log(value); // logs 'apple', 'banana', 'cherry'

}