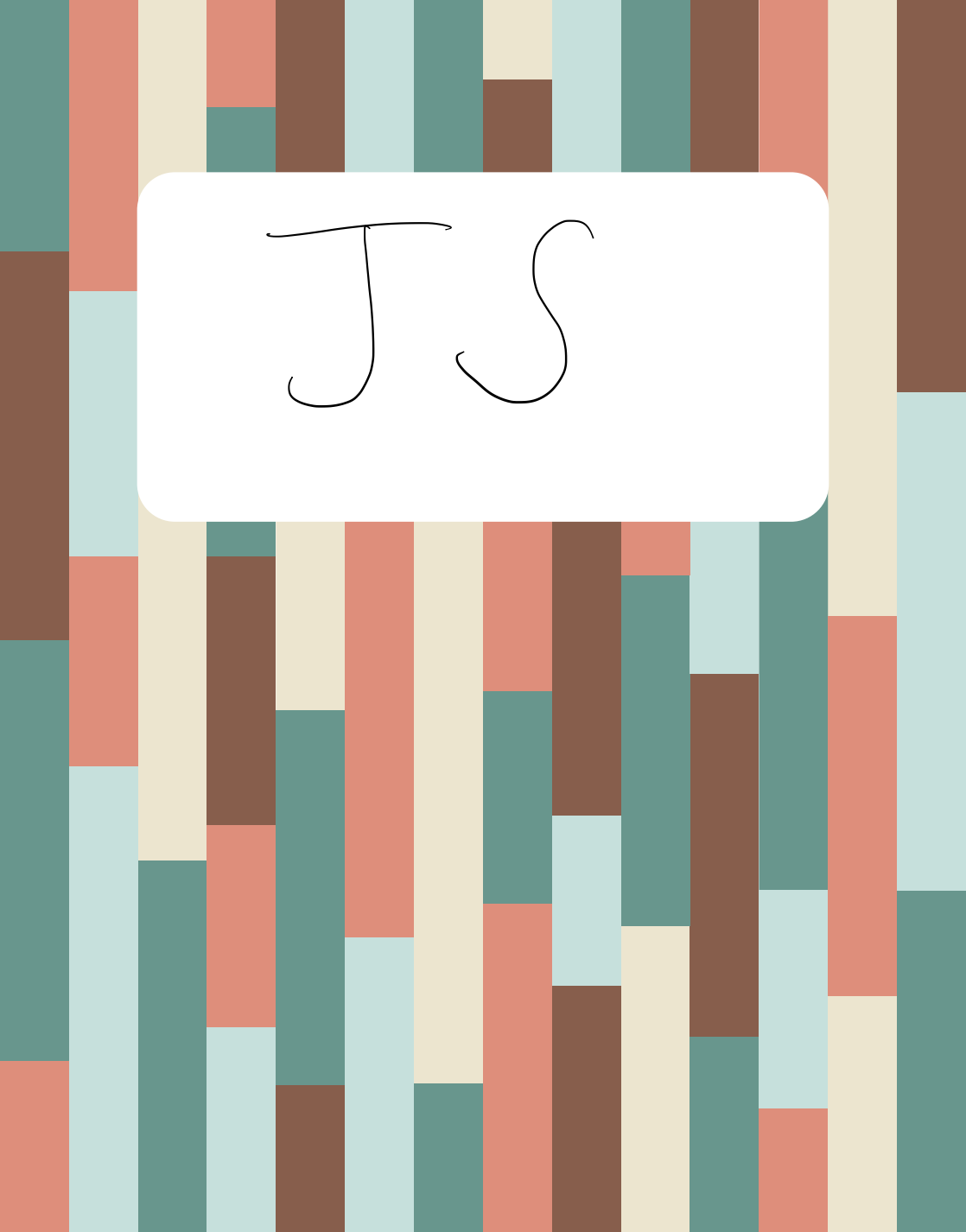

JS

JAVASCRIPT

Array Method

push & pop
- push: add last item
- pop: remove last item

shift & unshift
- shift: Remove 1st
- unshift: Add 1st

indexOf: return the 1st index at which a given element can be found.
return -1 if element is not present

slice: copy & return part of the array
- slice (a, b)
- slice ( ) → Copy all

filter ( ) – return an array filled with element that passes a test

map ( ) – return a new array in which each element is the result of a passed in function.

find ( ) – return the 1st element in an array that pass a test.

reduce ( ) – reduce the array to a single value, by executing a function on each element of the array

Destructuring

```
const {a, b} = c
```

- {a, b} → Create 2 variable & get the corresponding data in c
- c → An object

const [a, b] = [... array ...]

↳ [a gets 1st item.
  [b ____ 2nd item

PROTOTYPES

⊛ Constructor Function

```
function Student() {
    this.name = ""
    this.age = ""
}

var chinh = new Student()
```

→ Constructor Function.

→ Mỗi constructor Function. đc tạo ra. thì có sẵn một property là prototype

→ Có thể dùng prototype để tạo thêm hàm & biến và chúng sẽ được share bởi mọi object tạo ra. từ constructor Function

HIGHER ORDER FUNCTION

- 1 Function với khả năng return 1 Function
- Function luôn dùng khi return thứ khi return 1 Function
- Child Function can access data of parent function.

CLOSURE ⟶ Stateful Function.

- 1 dạng higher order Function mà khi gán nó vào 1 biến thì biến đó ∂ có khả năng truy xuất trực tiếp data mà phải thông qua cái Function con. Cái Function con có thể truy xuất data của cho.

Constructor.

→ Initialize new object by "new" keyword.
→ Need to use "this"
→ Stateful function.

Factory

→ Simply return a new object
→ No "this"
→ More private, only those has been returned. can be access outside closure

ASYNCHRONOUS

- All asynchronous task will be automaticlly consider as Macro Task and will be pushed to the back of the call stack until all the micro task finish executing

E.g: setTimeout([ ], 0) will not run. (even with 0 sec) until all the micro task is done

```
setTimeout(() => {
    console.log("Hello")
}, 0)
console.log("Hello after timeout")

//Result
//Hello after timeout
//Hello -> this appear later
```

→ console.log appears 1st as it is a micro task

- Trong 1 stack đang chờ để xử lí các asynchronous task, the smaller task will run 1st & then the bigger/heavier task

Engines

V8, spider....

Feature
- Single thread.
- Blocking
- Synchronous
- Workers

- Heap
- Call stack.

Environment.

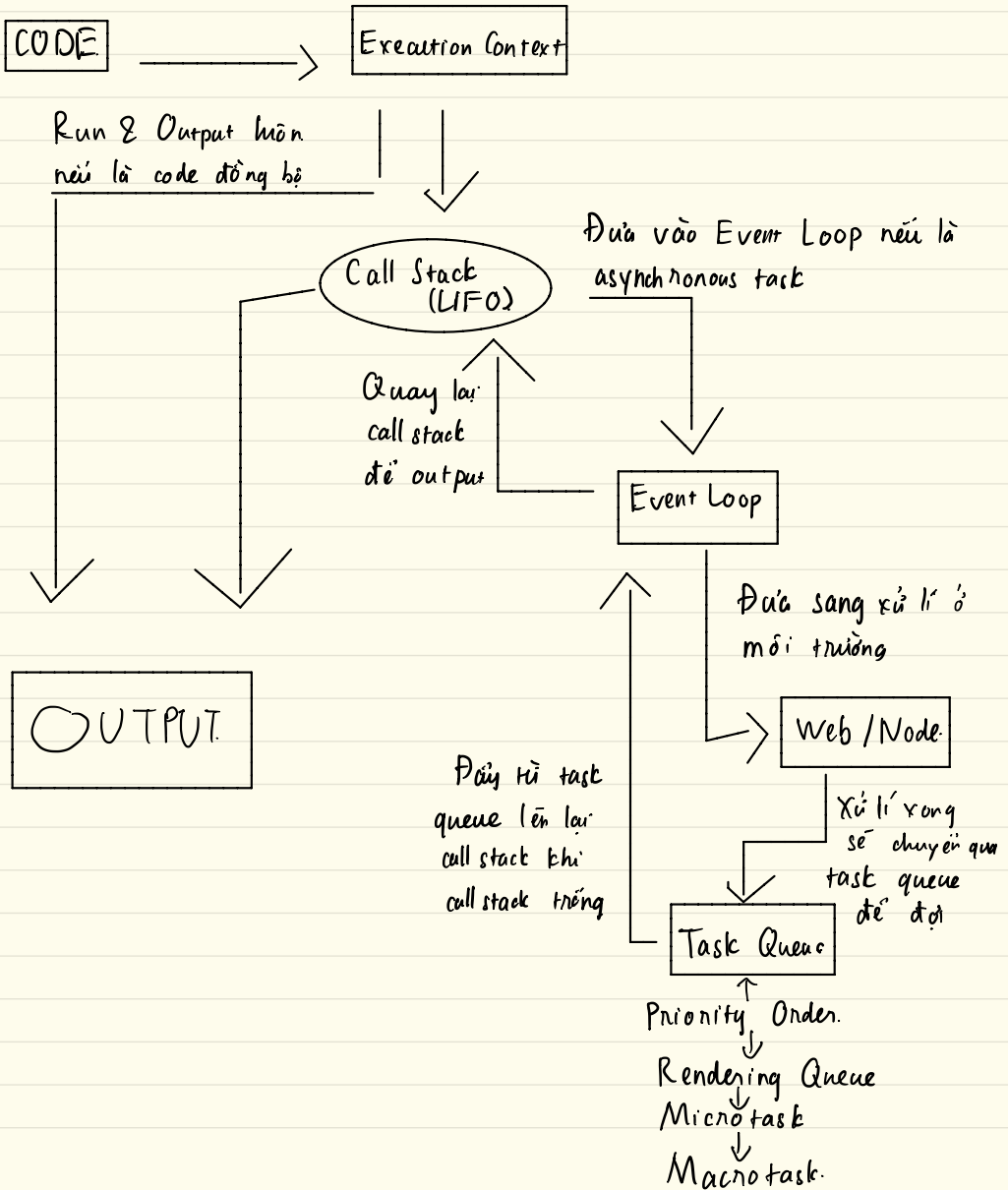
Browser & NodeJS

- Event Loop
- Task Queue
- Web/Node API

JS

1. Single / Multiple Thread.
2. Blocking & Non block
3. Async & Sync

→ Bản thân JS ko chạy đc đa luồng nhưng nhờ vào môi trường mà chạy đc asynchronous.



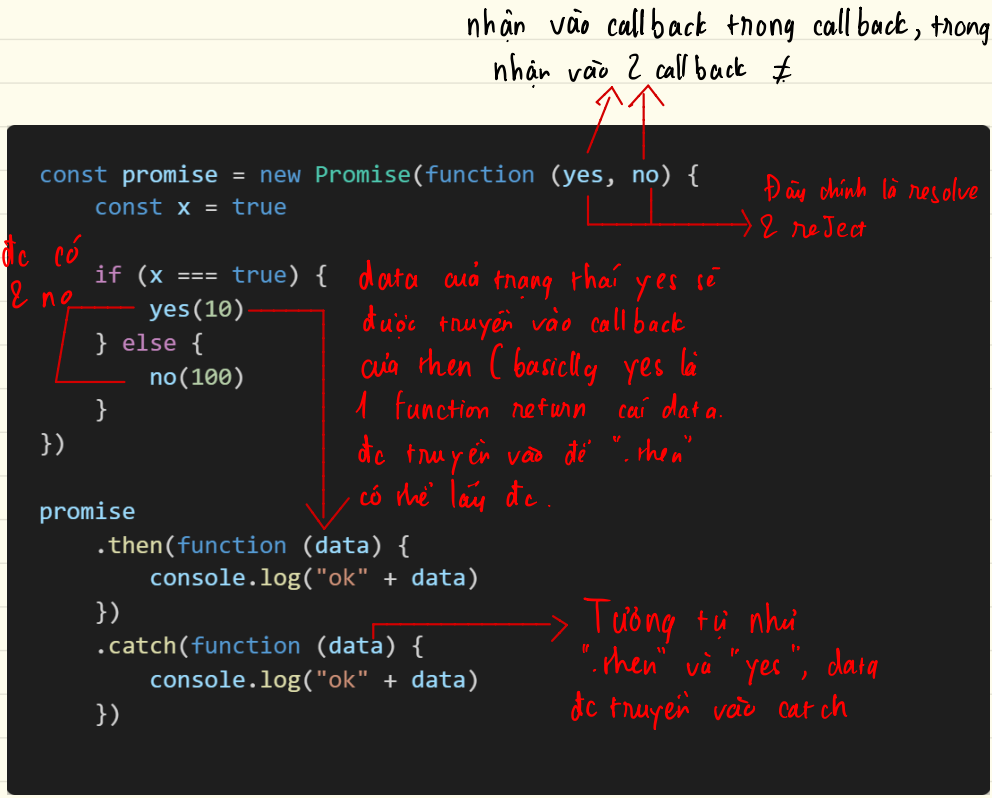
PROMISE

- Promise là các micro task, giúp điều khiển thứ tự xuất hiện / hoạt động của code.
  - Resolve
  - ReJect
  - Thực chất là 1 constructor Function.

- Có 2 dạng method.
  - ⊕ Method Prototype: Method đc gán chung cho mọi obJect
  - ⊕ Static Method: Method gắn thẳng vào class & đc gọi trực tiếp từ class chứ ko gọi từ obJect như prototype

→ ObJect được tạo bằng "new Promise" có thể truy cập được "then", "catch", "finally" (prototype)

→ Promise static method: resolve C); reJect C); all C)



, then(), .catch() là của object tạo từ new nên tùy ý sử dụng

- Nếu mà throw error thì phải catch
- Nếu dùng .then() liền nhau thì then() trước phải return data nếu muốn then sau có data

→ Then sẽ trả về 1 promise đã được resolve với data là data đã được return.

- Nếu return 1 hàm trong then thì trong hàm cũng phải return hàm

⊛ Nếu dùng các request function mà sẽ return Promise phụ thuộc vào nhau thì có thể return cái request

```
fetch()
    .then()
    .then(() => {
        console.log("Inside then")

        return fetch()
    })
    .then()
```

- ko return → then() kế tiếp ko nhận đc data.
- có return → return data → Trong callback ko thực hiện tính toán gì, then() kế nhận đc data

→ return hàm
- Truyền tên → Hàm tách riêng phải có return.
- Khai báo trực → Có return argument rõ ràng hơn
- Dùng arrow function.

- Các method prototype & static method method đều trả về 1 promise tức là ta có thể dùng then

- Phân biệt việc viết promise và dùng các API có sẵn trả về promise

ARROW FUNCTION

- Arrow function does not have a "this" context
  → "this" will refer to the value at the location where the function was defined
- If arrow function is a function statement, it should have curly bracket

→ In React, if we want to pass a function from parent to child, and call that in child with some argument, we have to call it via arrow function.

```
<Parent Func={Foo} />

<Child onClick={() => Foo(a)} />
```

ERROR HANDLING

Try ... Catch ... Finally – "Try" execute the code in the block, if it has any error will immediately go to the block of of code in nearest catch.

```
try {
    console.log("Inside try")
} catch {
    console.log("Inside catch")
} finally {
    console.log("This will happen no matter what")
}
```

- The error has to be system error, not syntax error
- The block inside finally will run after every thing above.

ERROR OBJ

- An autogenerated error object.
- Property { name: "Err name"
  message: "Text msg" }

→ Error object can be passed as an argument to catch.

→ For built-in Error object, name is the name of the error obj, the message is taken from the constructor.

→ We can create our own error object, make sure it has name & message attribute.

THROW

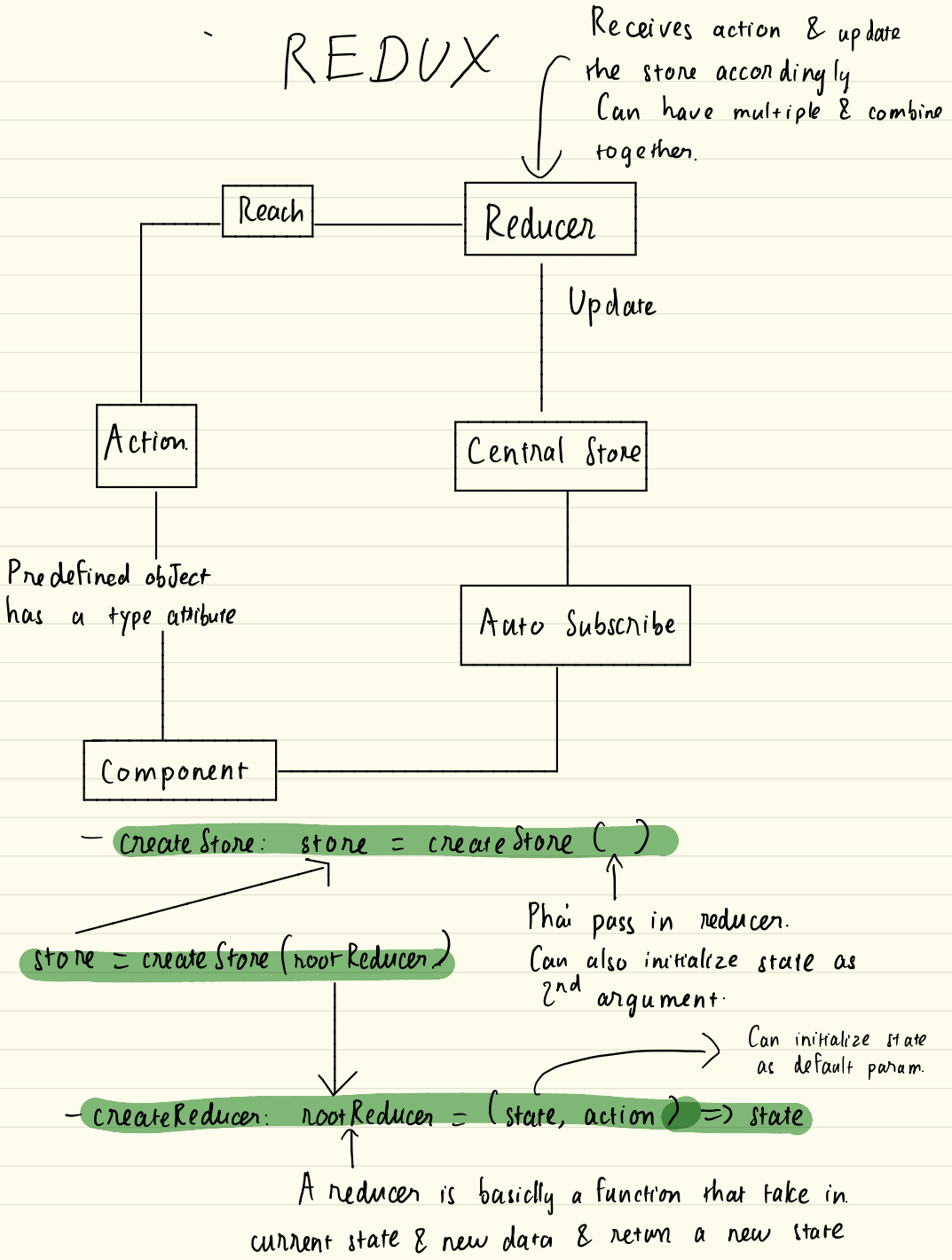
→ Generate an error that will be handle by catch

```
try {
    console.log("Inside try")

    throw new Error()
} catch {
    console.log("Inside catch")
}
```

Program will pass the error to catch as argument

REDUX


Receives action & update the store accordingly
Can have multiple & combine together.
Reducer
Reach
Update
Action
Central Store
Predefined object has a type attribute
Auto Subscribe
Component
- Create Store: store = createStore( )
Phải pass in reducer.
Can also initialize state as 2nd argument.
store = createStore (rootReducer)
Can initialize state as default param.
- createReducer: rootReducer = (state, action) => state
A reducer is basically a function that take in current state & new data & return a new state

- Action is a JS object that hold new data for the store. In production, we will dispatch the action and it will go through the reducer to decide which update will be done before updating the store with new data come along with the action.

REACT

HOOK

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

- Only call hook at the top level (❌ inside loop, condition or nested function)
- Only call hook from React Function

- If we update the state based on previous state, use a 2nd form of setState & pass in a function instead

```
setState(prevState => { count: prevState.count + 1 })
```

Each component has several "life-cycle" method that we can override
There are 3 phase in the life cycle

1. **Mounting**: the instance the component is created & inserted into the DOM
   constructor() → render() → componentDidMount()

2. **Updating**: can be caused by changes to props or state
   shouldComponentUpdate() → render() → componentDidUpdate()

3. **Unmounting**: called when a component is being removed from the DOM
   componentWillUnmount()

4. Error Handling: called when there is error during rendering, life-cycle or in constructor of any child component
   componentDidCatch()

constructor(props) → Called before it is mounted
Used to initializing local state & bind event handler.

render() → The only required method in class component

→ When called, examine this.props & this.state and return
- + React Elements
- + Arrays & Fragment
- + Portals
- + String & Num
- + Boolean / Null

- The render() function should be pure (& modify state, & interact directly with the browser)

componentDidMount() → Invoke immediatelly after the component is mounted
Intialization that requires DOM should go here
Good place to intiate network request to load data from remote endpoint.

componentDidUpdate() → Invoke immediately after updating occur
Good opportunity to opperate on DOM when the component has been updated.
Also a good place to do network request as long as we compare the current props to the previous props.

```
componentDidUpdate(prevProps) {
    if (this.props.userId !== prevProps.userId
        { this.fetchData(this.props.userId) }
}
```

componentWillUnmount() → Invoke immediately before a component is unmounted.
Perform any necessary clean up or cleaning up any subscription in componentDidMount
Should not call setState

useEffect - lets you perform side effect in functional component
- close replacement of componentDidMount, componentDidUpdate, componentWillUnmount
- Runs after every render
- The 2nd parameter is the list of dependencies that useEffect uses to know when to execute (componentDidUpdate)

useEffect(() => {}, [dependes])

↳ If it is an emty array, we mimic componentDidMount & call useEffect only once in initial render.

- To mimic componentWillUnmount, in the function pass in, we simply return a clean up function

```
useEffect(() => {
    console.log("Inside useEffect")

    return () => {
        console.log("This is clean up function")
    }
}, [])
```

→ This is the clean up function.

CONTEXT

- Provides a way to pass data through the component tree without having to pass props down manually at every level.
- To use context, we 1st create the context and then wrap the components tree and pass in the initial value for it.

<UserContext.Provider value={}><App /></...Provider>

- To use the context via hook, we 1st import it in our component
  import { useContext }
- → The useContext hook return the value of the context so we can assign it to a variable

useReducer() → A hook used for state management

reduce in JS	useReducer in React
array.reduce(reducer, initialValue)	useReducer(reducer, intialState)
SingleValue = reducer(accumulator, itemValue)	newState = reducer(currentState, action)
return a single value	returns a pair of value [newState, dispatch]

```
const initialValue = 0
const reducer = (state, action) => {
    switch (action.type) {
        case "inc":
            return state + 1
        case "dec":
            return  state - 1
        case "res":
            return initialValue
        default:
            return state
    }
}

const [count, dispatch] = useReducer(reducer, initialValue)

<button onClick={() => dispatch("inc")} />
<button onClick={() => dispatch("dec")} />
<button onClick={() => dispatch("res")} />
```

→ dispatch is a method to change the the state returned by useReducer, similar to setState

```
const initialValue = {
    firstCounter: 0,
}

const reducer = (state, action) => {
    switch (action.type) {
        case "inc":
            return { firstCounter: state + 1 }
        case "dec":
            return { firstCounter: state - 1 }
        case "res":
            return initialValue
        default:
            return state
    }
}

const [count, dispatch] = useReducer(reducer, initialValue)

<button onClick={() => dispatch({type: "inc"})} />
<button onClick={() => dispatch({type: "dec"})} />
<button onClick={() => dispatch({type: "res"})} />
```

> We can change the state to object, so the reducer can do so much more by switching through type in action & perform accordingly with the payload

If we have multiple state property & only want to change one, we can utilize the spread operator (still similar to useState & setState)

- To manage global state (like Redux), we can combine useReducer + useContext

```jsx
export const CountContext = React.createContext()

const intialValue = 0

const reducer = (state, action) => {
    switch (action) {
        case "inc":
            return state + 1
        case "dec":
            return state - 1
        case "res":
            return intialValue
        default:
            return state
    }
}

function App() {
    const [count, dispatch] = useReducer(reducer, intialValue)

    return (
        <CountContext.Provider
            value={{ countState: count, countDispatch: dispatch }}
        >
            <ComponentA />
        </CountContext.Provider>
    )
}

function ComponentA() {
    const countContext = useContext(CountContext)

    return (
        <div>
            <button onClick={() => countContext.countDispatch("inc")} />
            <button onClick={() => countContext.countDispatch("dec")} />
            <button onClick={() => countContext.countDispatch("res")} />
        </div>
    )
}
```

→ We can use Reducer as alternative for useState for data fetching

```
const initialState = {
    loading: true,
    error: "",
    post: {},
}

const reducer = (state, action) => {
    switch (action.type) {
        case "FETCH_SUCCESS":
            return {
                loading: false,
                post: action.payload,
                error: "",
            }
        case "FETCH_ERROR":
            return {
                loading: false,
                post: {},
                error: "Something went wrong",
            }
        default:
            return state
    }
}

function DataFetching() {
    const [state, dispatch] = useReducer(reducer, initialState)

    useEffect(() => {
        axios.get(...url...)
        .then(response => {dispatch({type:"FETCH_SUCCESS", payload:
response.data})})
        .catch(error => {dispatch({type:"FETCH_ERROR"})})

    }, [])

    return (
        <div>
            {state.loading ?  "Loading" : state.post.title}
            {state.error ? state.error : null}

        </div>
    )
}
```

Pure Components

Regular Component	Pure Components
- Does not implement shouldComponentUpdate, always return true	- Implements shouldComponentUpdate with a shallow props & state comparison. + SC prevState with currentState prevProps with currentProps

Shallow Comparison (SC)

- Primitive: return true if a & b have the same value & the same type
- Complex: return true if a & b reference the exact same object

```
function MemoComp({ name }) {
    return <div>{name}</div>
}

export default React.memo(MemoComp)
```

Refs — make it possible to access DOM node directly within React

```
class RefDemo extends Component {
    constructor(props) {
        super(props)
        this.inputRef = React.createRef()
    }

    componentDidMount() {
        this.inputRef.current.focus()
    }

    render() {
        return (
            <div>
                <input type="text" ref={this.inputRef} />
            </div>
        )
    }
}
```

→ We 1st need to create the Ref using React

→ Call the event specific to that ref

→ Use the ref attribute to assign the ref

→ We can also use ref on React class component.

```
class Input extends Component {
    constructor(props) {
        super(props)
        this.inputRef = React.createRef()
    }

    focusInput() {
        this.inputRef.current.focus()
    }

    render() {
        return (
            <div>
                <input type="text" ref={this.inputRef} />
            </div>
        )
    }
}

class FocusInput extends Component {
    constructor(props) {
        super(props)
        this.componentRef = React.createRef()
    }

    clickHandler = () => {
        this.componentRef.current.focusInput()
    }

    render() {
        return (
            <div>
                <Input ref={this.componentRef} />
            </div>
        )
    }
}
```

This has to match the function of child component

- We can also create the in parent component and forward it to the child (rarely use)

```
const Input = React.forwardRef((props, ref) => {
    return <input type="text" ref={ref} />
})

class FocusInput extends Component {
    constructor(props) {
        super(props)
        this.componentRef = React.createRef()
    }

    clickHandler = () => {
        this.componentRef.current.focus()
    }

    render() {
        return (
            <div>
                <Input ref={this.componentRef} />
                <button onClick={this.clickHandler} />
            </div>
        )
    }
}
```

Create child component as functional component & wrap in forwardRef & refer to the ref attribute

We can use the focus() directly since we are forwarding the ref

→ To use ref in Functional component, use Ref hook

```
function FocusInput() {
    const inputRef = useRef(null)
    useEffect(() => {
        inputRef.current.focus()
    }, [])
    return (
        <div>
            <input ref={inputRef} type="text" />
        </div>
    )
}
```

```
const refContainer = useRef(initialValue)
```

- Another use for the useRef hook

→ useRef() returns a mutable ref object whose .current is initialized to the passed argument. The returned object will persist for the full lifetime of the component

```
function HookTimer() {
    const [timer, setTimer] = useState(0)
    const interValRef = useRef()
    useEffect(() => {
        interValRef.current = setInterval(() => {
            setTimer((timer) => timer + 1)
        }, 1000)
        return () => {
            clearInterval(interValRef.current)
        }
    }, [])
    return (
        <div>
            HookTimer - {timer} -
            <button onClick={() => clearInterval(interValRef.
current)}>
                Clear Timer
            </button>
        </div>
    )
}
```

→ initialize the ref object

→ mutable JS object as we can change it directly (the difference is useRef will give the same ref object every render & will not cause re-render

→ can use the ref like plain JS object.

Use Callback()

- React auto re-render component whenever it detects any change in state or props, And when a parent component re-render → A child also re-render
- There's a scenario when we pass down a function to a child, and then the parent component re-render by the effect of state change → All the child component also re-render even if there's no change related to its state

→ This is because when the parent component re-render, all the function is also re-render even though it's not relevant to the change.

↳ Solution: use Callback + React.memo

- useCallback is a hook that return a memoized version of the callback function that only changes if one of the dependencies has changed.
  - It is useful when passing callbacks to optimized child components that rely on reference quality to prevent unnecessary render

```
function ParentComponent() {
    const [age, setAge] = useState(25)
    const [salary, setSalary] = useState(50000)

    const incrementAge = useCallback(() => {
        setAge(age + 1)
    }, [age])

    const incrementSalary = useCallback(() => {
        setSalary(salary + 1000)
    }, [salary])

    return (
        <div>
            <Title />
            <Count text="Age" count={age} />
            <Button handleClick={incrementAge}>Increment Age</
Button>
            <Count text="Salary" count={salary} />
            <Button handleClick={incrementSalary}>
Increment Salary</Button>
        </div>
    )
}
```

Only changes when age changes

Only changes when salary changes

```
function Title() {
    console.log("Rendering Title")
    return <h2>useCallback Hook</h2>
}

export default React.memo(Title)
```

```
function Button({ handleClick, children }) {
    console.log("Rendering button - ", children)
    return <button onClick={handleClick}>{children}</button>
}

export default React.memo(Button)
```

```
function Count({ text, count }) {
    console.log(`Rendering ${text}`)
    return (
        <div>
            {text} - {count}
        </div>
    )
}

export default React.memo(Count)
```

Implements pure component

Custom Hook

- A JS function whose name starts with "use"
- Can also call other hook if required.

↳ Share logic - Alternative to HOCs & Render Props

```
function useDocumentTitle(count) {
    useEffect(() => {
        document.title = `Count ${count}`
    }, [count])
}
```

```
function DocTitleOne() {
    const [count, setCount] = useState(0)
    useDocumentTitle(count)

    return (
        <div>
            <button onClick={() => setCount(count + 1)}>
                Count - {count}
            </button>
        </div>
    )
}
```

Simple Custom Hook

```
function useCounter(initialCount = 0, value) {
    const [count, setCount] = useState(initialCount)

    const increment = () => {
        setCount((prevCount) => prevCount + value)
    }

    const decrement = () => {
        setCount((prevCount) => prevCount - value)
    }

    const reset = () => {
        setCount(initialCount)
    }
    return [count, increment, decrement, reset]
}
```

```
function CounterOne() {
    const [count, increment, decrement, reset] = useCounter(0, 1)

    return (
        <div>
            <h2>Count = {count}</h2>
            <button onClick={increment}>Increment</button>
            <button onClick={decrement}>Decrement</button>
            <button onClick={reset}>Reset</button>
        </div>
    )
}
```

More complex custom hook

```
function useInput(initialValue) {
    const [value, setValue] = useState(initialValue)
    const reset = () => {
        setValue("")
    }
    const bind = {
        value,
        onChange: (e) => {
            setValue(e.target.value)
        },
    }
    return [value, bind, reset]
}
```

```
function UserForm() {
    const [firstName, bindFirstName, resetFirstName] = useInput(
"")
    const [lastName, bindLastName, resetLastName] = useInput("")

    const submitHandler = (e) => {
        e.preventDefault()
        alert(`Hello ${firstName} ${lastName}`)
        resetFirstName()
        resetLastName()
    }
    return (
        <div>
            <form onSubmit={submitHandler}>
                <div>
                    <label>First Name</label>
                    <input type="text" {...bindFirstName} />
                </div>
                <div>
                    <label>Last Name</label>
                    <input type="text" {...bindLastName} />
                </div>
                <button>Submit</button>
            </form>
        </div>
    )
}
```

More practical & complex custom hook

NODEJS

ERROR HANDLING

- Synchronous: [throw new Error()
  return next(new Error)

- Asynchronous: → return next(new Error)