# **Java Script ES6**

Refer the best module – [ES6 by JS](https://www.javascripttutorial.net/es6/)

### **Basics**

* [let](https://www.javascripttutorial.net/es6/javascript-let/) – declare block-scoped variables using the let keyword. With in {} braces scoped.
* [const](https://www.javascripttutorial.net/es6/javascript-let/) – declare block-scoped variables using the const keyword. But the value can’t be changed but returns new variable with value.
* var – declare global-scoped when declare outside of the function or block {}, can be used though out the class anywhere.

typesof(var) -can returns the type of an element. Instanceof

Arrow functions const fun = (…args)=>{}; here this couldn’t work.

const a = [1,2,3,4];

.map is used to return the value in object/array >> console.log(array.map(ar => ar+1));//[2,3,4,5,]

.filter is used to return the filtered object values >> console.log(a.filter(ar => (ar/2 ===1)));//2

Const date = new Date();

date.getFullYear(); date.getDay(); date.getMonth()+1;

let current date = `${date}-${month}-${year}` //String literals

Tip01: The advantage of using static functions or properties in a class is that:

* They can be used to create functions/properties which need not be present in the instances. This helps to maintain some isolation in the codebase.
* They reduce code redundancy in some cases.

Tip02: to iterate though the array need to use “of” like, for I in array. To iterate thought the object need to use “in” for I in object.

### **Class and Objects**

Classes were introduced in [ECMAScript 2015](https://262.ecma-international.org/6.0/) (ES6) to provide a cleaner way to follow object-oriented programming patterns. Class is blueprint of an object.

Classes are developed on behalf of constructor functions. In this type of implementation its hard to add functions. Like need to use Pen.prototype.showprice = ()=>

function Pen(name, color, price) {

this.name = name;

this.color = color;

this.price = price;

}

const pen1 = new Pen("Marker", "Blue", "$3");

console.log(pen1);

We can re-create the above example with the help of the class keyword. Have a look at the below code:

class Pen {

constructor(name, color, price){

this.name = name;

this.color = color;

this.price = price;

}

showPrice(){

console.log(`Price of ${this.name} is ${this.price}`);

}

}

const pen1 = new Pen("Marker", "Blue", "$3");

pen1.showPrice();

Implement JavaScript inheritance using **extends** and **super – here mostly can use static types.**

Objects: Objects in everyday life have properties and “method” actions. Take, for instance, a fan. It's an object with humidity.

Creating an object without class:

const person = {

‘name’:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

Console.log(Person.name);

For key properties– dot

To Access a property > person.age;

To Delete a property> delete person.age;

To Modify a property or adding property(if new)> person.age = 30;

For without key properties– it is achievable by square brackets: To Access a property > person[‘name’];

Object Spread Operator

The object spread operator is a popular and powerful syntax in JavaScript. The spread operator takes all the key-value pairs of an object and copies the key name and value into a new object.

Object Destructuring

Object destructuring is an important feature in JavaScript that allows you to pull out values from an object and assign them to individual variables.

const person = { name: 'lawal', age: 39 };

const { person, age } = person;

console.log(name); // 'lawal'

console.log(age); // 39

### **Arrays**

It stores multiple values and elements in **one** variable. These values can be of any data type — meaning you can store a string, number, boolean, and other data types in one variable.

There are two standard ways to declare an array in JavaScript. These are either via the array constructor or the literal notation and using new Array.

// Using array constructor

let array = new array("John Doe", 24, true);

// Using the literal notation

let array = ["John Doe", 24, true];

const cars = new Array("Saab", "Volvo", "BMW");

Accessing an element => array[0]

Changing an element => array[0] = 10;

Array to String => array.to String();

array.length; array.sort();

adding an element: deleting an element  
array.push(45); -adds at end array.pop() – remove at end

array.unshift(77 ); - adds at beginning array.shift() – removes at beginning

array1.concat(array);

Splicing and Slicing Arrays

fruits.splice(2, 0, "Lemon", "Kiwi");- The splice() method can be used to add new items to an array.

const citrus = fruits.slice(1); - The slice() method slices out a piece of an array into a new array.

Array Destructuring: (… rest parameter)

function getScores() {

return [70, 80, 90, 100];

}

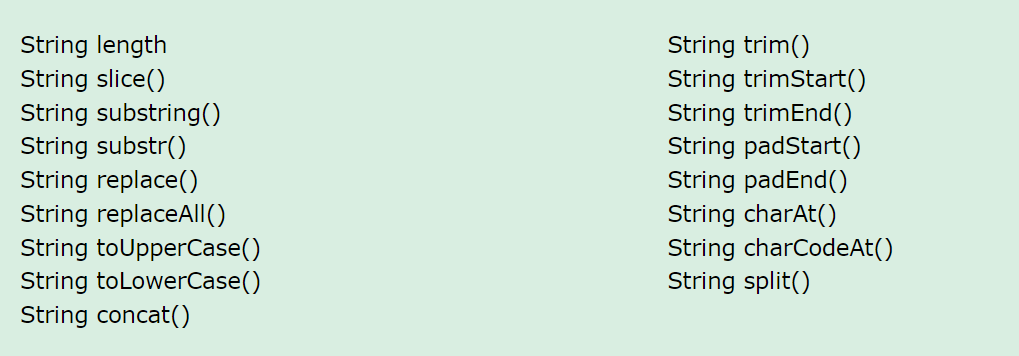
let [x, y, z] = getScores();

console.log(x); // 70

console.log(y); // 80

console.log(z); // 90

### **Strings**



### **Collections**

Map Object - the Map type that holds a collection of key-value pairs.

1. An object always has a default key like the [prototype](https://www.javascripttutorial.net/javascript-prototype/).
2. A key of an object must be a [string](https://www.javascripttutorial.net/javascript-string/) or a [symbol](https://www.javascripttutorial.net/es6/symbol/), you cannot use an object as a key.
3. An object does not have a property that represents the size of the map.

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber']

]);

Key is use din map methods=>map.set(john:’admin’), map.has(john);map.delete(john); map.clear()

let john = { name: 'John Doe' },

lily = { name: 'Lily Bush' },

peter = { name: 'Peter Drucker' };

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber'],

]);

for (const user of userRoles.keys()) { //can be used .values() {value to be returned} instead of .keys() or .entries {}

console.log(user.name);

}

for (let role of userRoles.values()) {

console.log(role);

}

for (const role of userRoles.entries()) {

console.log(`${role[0].name}: ${role[1]}`);

}

for (let [key, value] of chars.entries()) {

console.log(key === value);

}

chars.forEach(role => console.log(role.toUpperCase()));

Here are the main difference between a Map and a WeekMap:

* Elements of a WeakMap cannot be iterated.
* Cannot clear all elements at once.
* Cannot check the size of a WeakMap.

Set Object - the Set type that holds a collection of unique values.

let setObject = new Set();

let chars = new Set(['a', 'a', 'b', 'c', 'c']);

let size = chars.size;

chars.add('d');

let exist = chars.has('a');

console.log(exist);// true

chars.delete('d');

chars.clear();

for (let role of chars) {

console.log(role);

}

WeakSets

A WeakSet is similar to a Set except that it contains only objects. Since objects in a WeakSet may be automatically garbage-collected, a WeakSet does not have size property. Like a WeakMap, you cannot iterate elements of a WeakSet, therefore, you will find that WeakSet is rarely used in practice. In fact, you only use a WeakSet to check if a specified value is in the set.

# **React JS**

Refer the Git Hub repository - [**React\_Expenses\_Project**](https://github.com/sudhaScode/React_Expenses_Project)

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

### **Getting started with React**

**==========================================================**

To setup client by learning below commands

$mkdir client

$npx create-react-app expenses

$npm install {all dependencies like react, react-dom, and react-scripts, webpack webpack-dev-server webpack-cli --save}

$cd expenses

$npm start

===================================================================

setup server

$npm init -y

$npm install {all server dependencies like express cors nodemon}

$npm start

**==========================================================**

### **Components in React**

### **State, props, and Events**

* State management
* Components splitting and props
* Returning components conditionally

To send data from Parent to child can use JS variables inside the {} (JSX) at component calling**. [calling parent variable inside child component]**

To send data from child to parent can use event Listener functions with props at component calling, then calling that function inside of the child with object parameter(**props.parentfun(childdata)**), then the parent uses the objects inside it. **. [calling parent function inside child component, which handle the data send to that from child]**

Dynamic component return use achieved by useState hook and with ternary or conditional statements. And using .map can update list of objects for a component dynamically.

Dynamic style updating for components is achieved by inline style {{}} or className = {`exp+${with JS variable}`}

Dynamic styling React components

* >> **npm install –save styled-**components {import styled from ‘styled-components’;} const button = styled.button`css bricks`; inside back ticks for nested elements use &
* **Inline styling**
* **CSS Modules –** className = {props.styles}

<https://github.com/academind/react-complete-guide-code/tree/06-styling>

**Debugging React apps:** use console are result of npm start or npm run watch

**Styling:**

Inline CSS : style ={{}}

CSS-Module : className = {classes.button} [App.module.css]

Custom Error Handling : based on the value of the state, rendering a component

const submitHandler =(event)=>{

        event.preventDefault();

        if(userName.length === 0 || userAge.length === 0){

            setError({

                title:"Invalid input",

                message:"Please enter a valid name and age (non-empty values).",

            });

            setUsername(null);

            setUserage(null);

        }

        else if(+userAge<1){

            setError({

                title:"Invalid age",

                message: "  please enter a valid age > 0",

            });

            setUsername(null);

            setUserage(null);

        }

        else{

        const userData ={

            name:userName,

            age: userAge,

        }

        props.onAdd(userData);

        setUsername('');

        setUserage('');

    }

    }

### **Fragments, Portals & Refs -Hooks**

useState hook updates the value of the variable in next re-render, where useRef make updates the value instantly without waiting for re-render it don’t cause to rerender – no display of update value – when just to read .

const usernameRef =useRef();

ref ={usernameRef}

enteredName = usernameRef.current.value;

React.forwardRef(( props, ref)

|  |  |
| --- | --- |
| **useState()** | **useReducer()** |
| * The main state management “tool”. * Great for independent pieces of state/data. * Great if state updates are easy and limited to a few kinds of updates. | * Great if you need to add more power. * Should be consider if you had related pieces of data/state. * Can be helpful if you have more complex state updates. |

<React.Fraagment>

</React.Fraagment>

const root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<React.StrictMode> <App/></React.StrictMode>);

<https://github.com/academind/react-complete-guide-code/tree/09-fragments-portals-refs/code>

sideEffects:

useEffect: useEffect ((()=>{ //The code will run here},[//dependency])); When component mounted at least the useEffect blocked to code will runs , if dependency is given useeffect listens the change of the array elements, as per the change useeffect re-renders the updated components.

To make a component responsive with side effects(user interacts), based on current state of component.

**Cleanup** function in useEffect , use return at code block.

useEffect(()=>{

const identifier = setTimeout(()=>{

console.log(‘checking form validity’);

setFormInvalid(enteredEmail.includes(‘@’) && enteredPassword.trim().length>6);

}, 500);

return ()=>{

cosole.log(‘clean up!’);

clearTimeout(identifier);

};

}, [enteredEmail, enteredPassword]);

**useReducer:** One reducer equal to two useState hook, only one reducer is enough to use&set and validate the state

cosnt [state, dispatchFn] = useReducer(resucerFn, initialSate, initFn);

useCallback: this hook is used to improve the performance of an component by saving the implementation and invoking the implementation based on the state changes of component variables(keeping as dependencies[]). Normally used for functional changes.

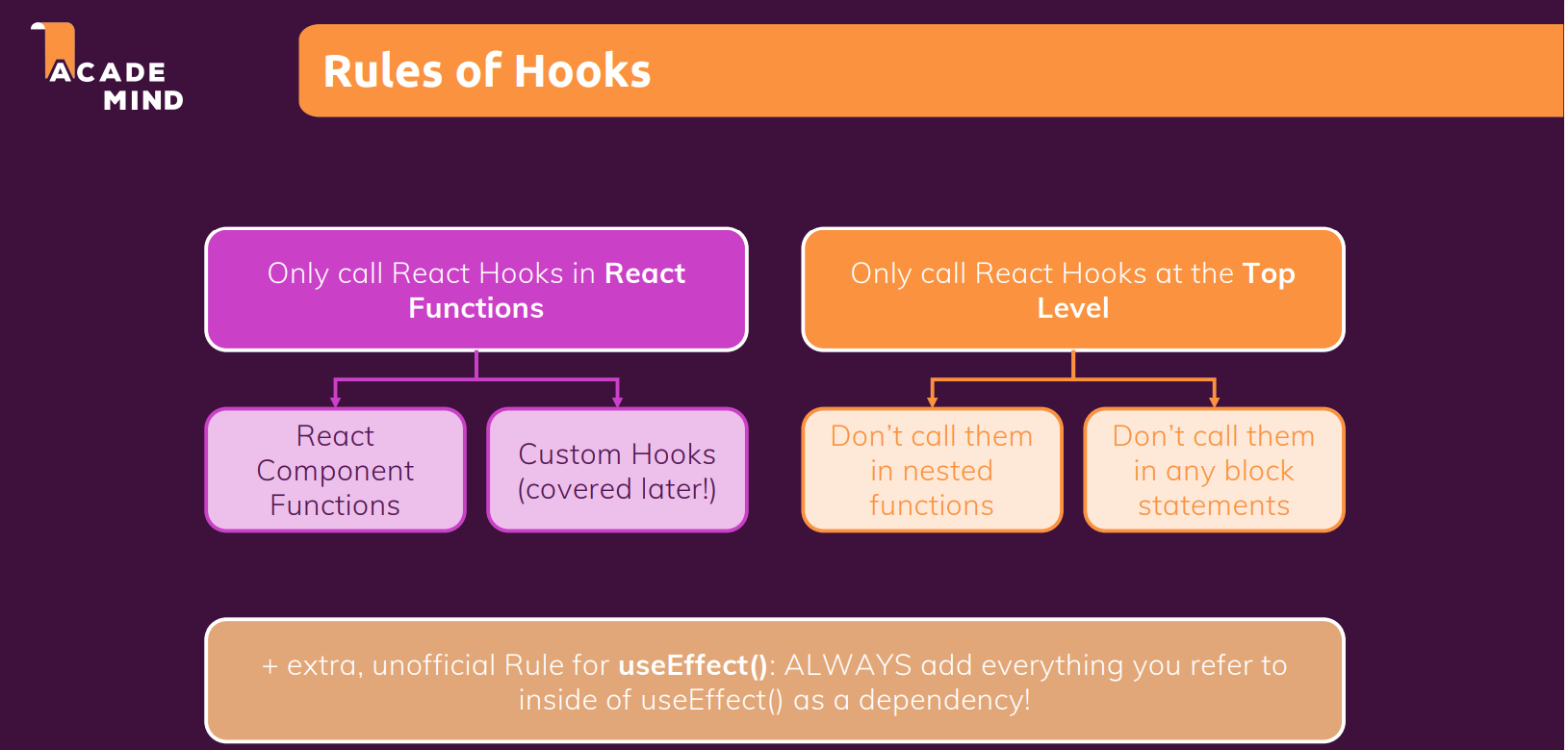
usecallback(()=>{},[]);

useMemo also a similar hook used to improve performance by optimizing complex calculation based on dependencies.

React.memo is used to trace the props changes send to the child component and if changes are made to the props values then the component re renders other wise it won’t allow to re render. It is used at exporting syntax. **export default React.memo(App);**

**React Context: Manage state (Shared state among componenets)**

**const ctx = useContext(**React.createContext());



<https://github.com/academind/react-complete-guide-code/tree/10-side-effects-reducers-context-api>

**Backdrop and overlay in React application:**

**Add overlay id in index.html at public folder.**

import styles from './Modal.module.css';

import { Fragment } from 'react';

import ReactDOM from 'react-dom';

const Backdrop =props=>{

        return <div className={styles.backdrop} onClick ={props.onClose}></div>

};

const ModalOverlay  = props =>{

    return(

        <div className={styles.modal}>

            <div className={styles.content}>{props.children}</div>

        </div>

    );

};

const portalElement = document.getElementById('overlays');

const Modal = props=>{

    return(

    <Fragment>

        {ReactDOM.createPortal(<Backdrop onClose={props.onClose}/>, portalElement)}

        {ReactDOM.createPortal(<ModalOverlay>{props.children}</ModalOverlay>, portalElement)}

    </Fragment>

    );

};

export default Modal;

Using context hook getting the props of distant component is availed by following process: (Food- Order project)

Follow Up 1: create context

const Cartcontext = React.createContext({

    items:[],

    totalAmount:0,

    addItem: (item) =>{},

    removeItem: (id)=>{},

})

This Cartcontext works as a wrapper element.

Follow Up 2: create a component for provider, here the prop objects are handled

import Cartcontext from "./cart-context"

import { useReducer } from "react";

const defaultCartState = {

    items:[],

    totalAmount:0,

};

const cartReducer = (state, action) =>{

    if (action.type === 'ADD'){

        const updatedItems = state.items.concat(action.item);

        const updatedTotalAmount = state.totalAmount + action.item.price\*action.item.amount;

        return{

            items:updatedItems,

            totalAmount:    updatedTotalAmount

        };

    }

    return defaultCartState;

};

const CartProvider = props=>{

    const   [cartSate, dispatchCartAction] =useReducer(cartReducer, defaultCartState);

    const addItemToCartHandler =(item) =>{

        dispatchCartAction({type:'ADD', item: item })

    };

    const removeItemFromCartHandler =(id) =>{

        dispatchCartAction({type: 'REMOVE', id: id})

    };

    const cartContext = {

        items:cartSate.items,

        totalAmount: cartSate.totalAmount,

        addItem: addItemToCartHandler,

        removeItem: removeItemFromCartHandler,

    };

    return(

        <Cartcontext.Provider value={cartContext}>

            {props.children}

        </Cartcontext.Provider>

    );

}

export default CartProvider;

Follow Up 3: Fast forwarding the user input by useRef

import classes from './Input.module.css';

import React from 'react';

const Input = React.forwardRef(( props, ref) =>{

    return(

        <div className={classes.input}>

            <label htmlFor={props.input.id}>

               {props.label}

            </label>

            <input ref ={ ref} id ={props.input.id} {...props.input}/>

        </div>

    );

});

export default Input;

Follow Up 4: get the user input from child to parent and update the context object:

import classes from './MealItem.module.css';

import MealItemForm from './MealItemForm';

import { useContext } from 'react';

import Cartcontext from '../../../store/cart-context';

const MealItem = props =>{

    const cartCtx = useContext(Cartcontext);

    const price = `$${props.price}`;

    const onAddtoCartHandler = amount =>{

        cartCtx.addItem({

            id: props.id,

            name: props.name,

            amount:amount,

            price: props.price

        });

    };

    return(

        <li className={classes.meal}>

            <div>

                <h3>{props.name}</h3>

                <div className={classes.description}> {props.description}</div>

                <div className={classes.price}>{price}</div>

            </div>

            <div>

                <MealItemForm id = {props.id} onAddtoCart ={onAddtoCartHandler}/>

            </div>

        </li>

    );

}

export default MealItem;

Follow Up 5: render the context property in current component where needs to render:

import { useContext } from "react";

import CartIcon from "../Cart/CartIcon";

import styles from './HeaderCartButton.module.css';

import Cartcontext from "../../store/cart-context";

const HeaderCartButton = props =>{

    const cartCtx = useContext(Cartcontext);

    const numberofCartItems = cartCtx.items.reduce((current,item)=>{

        return current+item.amount},0);

    return(

        <button className={styles.button} onClick ={props.onClick}>

            <span className={styles.icon}>

                <CartIcon/></span>

            <span>Your Cart</span>

            <span className={styles.badge}>{numberofCartItems}</span>

        </button>

    );

};

export default HeaderCartButton;