JavaScript

Basics (Fundamentals)

Variables in JavaScript are used to store data that can be referenced and manipulated in a program. JavaScript provides three ways to declare variables: `var`, `let`, and `const`. Each has different behavior in terms of scope, redeclaration, and mutability.

1. var (function-scoped, can be redeclared)

- function-scoped: the variable is accessible within the function where it is Declared.
- can be redeclared: you can declare the same variable multiple times within the same scope.
- hoisted to the top: the declaration is moved (hoisted) to the top of the scope but **not the initialization**.
- can be updated: the value of a `var` variable can be changed.

Example of `var`

```
var x = 10;

console.log(x); // 10

var x = 20; // redeclaration is allowed

console.log(x); // 20

if (true) {

   var y = 30; // accessible outside the block

}
```

```
console.log(y); // 30 (var is not block-scoped)
```

Hoisting example

```
console.log(a); // undefined (hoisting happens, but only
the declaration, not initialization)

var a = 5;

console.log(a); // 5
```

2. `let` (block-scoped, cannot be redeclared)

Introduced in ES6, `let` provides better scoping rules and prevents issues caused by `var`.

Key features of `let`

- block-scoped: the variable is accessible only within the block `{}` where it is declared.
- cannot be redeclared: you cannot declare the same variable within the same scope.
- hoisted, but not initialized: unlike `var`, accessing a `let` variable before declaration results in a `ReferenceError`.
- can be updated: you can change the value of a `let` variable.

Example of `let`

```
let x = 10;
console.log(x); // 10

// let x = 20; //error: cannot redeclare block-scoped
variable 'x'
x = 20; // allowed (updating value)
console.log(x); // 20

if (true) {
```

```
    let y = 30;
    console.log(y); // 30 (accessible within the block)
}
// console.log(y); //  error: y is not defined
(block-scoped)
```

Hoisting example

```
console.log(b); // ReferenceError: cannot access 'b' before
initialization
let b = 5;
console.log(b); // 5
```

3. const (block-scoped, immutable)

The const keyword is used to declare constants, meaning the variable cannot be reassigned after being initialized.

Key features of const

- block-scoped: like let, const is limited to the block {} where it is defined.
- cannot be redeclared: const cannot be redefined in the same scope.
- cannot be reassigned: once assigned, its value cannot be changed.
- hoisted, but not initialized: like let, accessing a const variable before declaration causes an error.

Example of const

```
const PI = 3.14;
console.log(PI); //3.14

// PI = 3.1415; //error: assignment to constant variable
```

```
// const PI = 3.1415; // error: cannot redeclare
block-scoped variable

if (true) {
    const gravity = 9.8;
    console.log(gravity); // 9.8
}
// console.log(gravity); // error: gravity is not defined
(block-scoped)
```

Hoisting example

```
console.log(c); //ReferenceError: cannot access 'c' before
initialization
const c = 5;
console.log(c); // 5
```

For objects and arrays in const

Although const prevents reassignment of the variable itself, it **does not** make objects and arrays immutable.

```
const person = { name: "John", age: 25 };
person.age = 30; // allowed (modifying object properties)
console.log(person.age); // 30

// person = { name: "Doe" }; // error: assignment to constant
variable
```

Data Types:

JavaScript has different types of values that we use to store and manipulate data. These values are categorized into two main types:

- Primitive Data Types (Simple values)
- Non-Primitive (Reference) Data Types (Complex values like objects)

1. Primitive Data Types (Stored directly in memory)

Primitive data types are the most basic types in JavaScript. They hold only a single value and are immutable (cannot be changed directly).

1. Number

- Used for all numeric values, including integers and decimals.
- JavaScript does not have separate types for integers and floats.

  Example:

```
let age = 25; // integer
let price = 99.99; // decimal
```

2. String

- Used for text (a sequence of characters).
- Must be enclosed in single ('), double ("), or backticks (``).

  Example:

```
let name = "John Doe"; // double quotes
let message = 'Hello, world!'; // single quotes
let template = `My name is ${name}`; // template
literals (ES6 feature)
```

3. Boolean

- Represents true or false values.
- Used in decision-making and conditions.

  Example:

```
let isJavaScriptFun = true;
let isRaining = false;
```

4. Undefined

- A variable is `undefined` if it has been declared but not assigned a value.

Example:

```
let x; // undefined
console.log(x); // Output: undefined
```

5. Null

- Represents intentional absence of any value.
- It is not the same as `undefined`.

Example:

```
let emptyValue = null;
console.log(emptyValue); // Output: null
```

6. Symbol (ES6 Feature)

- Used to create unique and immutable values.
- Mostly used as object property keys.

Example:

```
let symbol1 = Symbol('id');
let symbol2 = Symbol('id');
console.log(symbol1 === symbol2); // Output: false
(always unique)
```

7. BigInt (ES11 Feature)

- **Used for very large numbers beyond** `Number.MAX_SAFE_INTEGER`.

Example:

```
let bigNumber = 123456789012345678901234567890n;
console.log(bigNumber); // Output:
123456789012345678901234567890n
```

2. Non-Primitive (Reference) Data Types (Stored in memory as references)

Non-primitive data types store complex values and are mutable (can be changed).

1. Object

- A collection of key-value pairs.
- Used to store multiple values in a structured way.

  Example:

```
let person = {
    name: "John",
    age: 30,
    isStudent: false
};
console.log(person.name); // Output: John
```

2. Array

- A special type of object that holds multiple values in an ordered list.
- Uses indexing, starting from `0`.

  Example:

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits[1]); // Output: Banana
```

3. Function

- A reusable block of code that performs a specific task.

  Example:

```
function greet(name) {
    return `Hello, ${name}!`;
}
console.log(greet("Alice")); // Output: Hello, Alice!
```

4. Date

- Used to work with dates and times.

Example:

```
let today = new Date();
console.log(today); // Output: current date and time
```

5. Regular Expression (RegExp)

- Used for pattern matching and searching in strings.

  Example:

  ```
  let pattern = /hello/i; // Case-insensitive match for
  "hello"
  let result = pattern.test("Hello World");
  ```

- `console.log(result); // Output: true`

Operators (Used to perform operations on values.)

- : +, -, *, /, %, ++, --
- Comparison: == (checks value), === (checks value + type), !=, !==, <, >, <=, >=
- Logical: && (AND), || (OR), ! (NOT)
- Assignment: =, +=, -=, *=, /=
- Example:

  ```
  let a = 10, b = 5;

  console.log(a + b); // 15

  console.log(a === b); // false
  ```

```
console.log(a > 5 && b < 10); // true
```

Conditional Statements (Used for decision-making)

JavaScript provides the following conditional statements:

- `if` statement
- `if...else` statement
- `if...else if...else` statement
- `switch` statement
- Ternary operator (`? :`)

1. `if` Statement (Basic Condition Check)

The `if` statement executes a block of code only if a condition is `true`. If the condition is `false`, the code inside `if` will not run.

```
if (condition) {
    // Code to run if condition is true
}
```

```
let age = 20;
if (age >= 18) {
    console.log("You are eligible to vote.");
} // Output: You are eligible to vote.
```

If age is less than 18, nothing will be printed because the condition is `false`.

2. `if...else` Statement (Condition with Alternative Code)

The `if...else` statement provides an alternative block of code if the condition is `false`.

```
if (condition) {
    // Code runs if condition is true
} else {
    // Code runs if condition is false
}
```

```
let age = 16;
if (age >= 18) {
    console.log("You are eligible to vote.");
} else {
    console.log("You are not eligible to vote.");
}
// Output: You are not eligible to vote.
```

3. `if...else if...else` Statement (Multiple Conditions Check)

The `if...else if...else` statement allows checking multiple conditions. If the first condition is false, the next condition is checked, and so on. If none of the conditions are true, the `else` block is executed.

```
if (condition1) {
    // Code runs if condition1 is true
} else if (condition2) {
    // Code runs if condition1 is false and condition2 is true
} else {
    // Code runs if both conditions are false
}
let score = 75;
if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 80) {
    console.log("Grade: B");
```

```
} else if (score >= 70) {
    console.log("Grade: C");
} else {
    console.log("Grade: F");
}
// Output: Grade: C
```

4. `switch` Statement (Best for Multiple Choices)

The `switch` statement is used when we need to compare a single value against multiple possible values. It is often faster and cleaner than using multiple `if...else if` conditions.

```
switch (expression) {
    case value1:
        // Code runs if expression === value1
        break;
    case value2:
        // Code runs if expression === value2
        break;
    default:
```

```
        // Code runs if no cases match
}
```

Example:

```
let day = "Monday";
switch (day) {
    case "Monday":
        console.log("Start of the week!");
        break;
    case "Friday":
        console.log("Weekend is near!");
        break;
    case "Sunday":
        console.log("It's a holiday!");
        break;
    default:
        console.log("It's a normal day.");
}
```

```
// Output: Start of the week!
```

Important Notes about `switch`:

- Use `break` to stop execution after a match. Without `break`, JavaScript will continue executing the next cases (this is called fall-through).
- If no case matches, the `default` block is executed.

5.Ternary Operator (`? :`) – Shorter Way to Write `if...else`

The ternary operator is a shortcut for `if...else` statements. It is useful when you have a simple condition and want to return a value based on `true` or `false`.

```
condition ? value_if_true : value_if_false;
```

Example:

```
let age = 18;

let message = (age >= 18) ? "You can vote!" : "You cannot vote.";

console.log(message);

// Output: You can vote!
```

- If the condition (`age >= 18`) is true, `"You can vote!"` is returned.
- If the condition is false, `"You cannot vote."` is returned.

6. Nested `if` Statements (if Inside if)

An `if` statement inside another `if` statement is called a nested if.

Example:

```
let num = 10;

if (num > 0) {

    if (num % 2 === 0) {

        console.log("Positive even number");

    } else {

        console.log("Positive odd number");

    }

} else {

    console.log("Negative number or zero");

}

// Output: Positive even number
```

Loops

Loops in JavaScript allow us to execute a block of code multiple times until a certain condition is met. This makes programming more efficient, reducing the need to write repetitive code.

1.Why Use Loops?

Instead of writing the same code multiple times, we can use loops to repeat a task until a condition is satisfied.

Example Without Loop.

```
console.log("Hello, World!");

console.log("Hello, World!");

console.log("Hello, World!");

console.log("Hello, World!");

console.log("Hello, World!");
```

This is inefficient. Instead, we can use a loop:

Example With Loop:

```
for (let i = 0; i < 5; i++) {

    console.log("Hello, World!");  }
```

This prints "Hello, World!" five times but with much less code.

2.Types of Loops in JavaScript

JavaScript has different types of loops:

- for loop – Used when the number of iterations is known.
- while loop – Used when the number of iterations is unknown.

- do...while loop – Similar to while, but runs at least once.
- for...in loop – Loops over object properties.
- for...of loop – Loops over iterable objects like arrays.

3. for Loop (Fixed Iterations)

The for loop is used when we know how many times we need to run a block of code.

```
for (initialization; condition; update) {
    // Code to execute
}
```

- Initialization: Runs once before the loop starts.
- Condition: The loop runs as long as this is true.
- Update: Runs after each iteration (increases or decreases the counter).

```
for (let i = 1; i <= 5; i++) {
    console.log("Iteration:", i);
}
// Output:
// Iteration: 1
// Iteration: 2
// Iteration: 3
```

```
// Iteration: 4

// Iteration: 5
```

4. while Loop (Unknown Iterations)

The while loop is used when we don’t know in advance how many times the loop should run. It runs as long as the condition is true.

```
while (condition) {

    // Code to execute

}
```

Example:

```
let count = 1;

while (count <= 5) {

    console.log("Count:", count);

    Count++;

}

// Output:

// Count: 1

// Count: 2

// Count: 3

// Count: 4
```

```
// Count: 5
```

- The loop checks the condition first before running the block of code.
- If the condition is false, the loop does not execute at all.

5.do...while Loop (Runs At Least Once)

The do...while loop is similar to while, but it guarantees at least one execution before checking the condition.

```
do {
    // Code to execute
} while (condition);
```

Example:

```
let num = 1;
do {
    console.log("Number:", num);
    num++;
} while (num <= 5);
// Output:
// Number: 1
// Number: 2
// Number: 3
```

```
// Number: 4

// Number: 5
```

Even if num was already greater than 5, the loop would run once before checking the condition.

6. for...in Loop (Looping Over Objects)

The for...in loop is used to iterate over the properties (keys) of an object.

```
for (let key in object) {
    // Code to execute
}
```

Example:

```
let person = { name: "John", age: 30, city: "New York" }
for (let key in person) {
    console.log(key, ":", person[key]);
}
// Output:
// name : John
// age : 30
// city : New York
```

- key holds the property name, and person[key] gives the value.

7. for...of Loop (Looping Over Arrays & Strings)

The for...of loop is used to iterate over iterable objects like arrays, strings, maps, and sets.

```
for (let value of iterable) {
    // Code to execute
}
```

Example (Looping Over an Array):

```
let fruits = ["Apple", "Banana", "Cherry"];
for (let fruit of fruits) {
    console.log(fruit);
}
// Output:
// Apple
// Banana
// Cherry
```

Example (Looping Over a String):

```
let word = "JavaScript";
for (let letter of word) {
    console.log(letter);
```

```
}

// Output:

// J

// a

// v

// a

// S

// c

// r

// i

// p

// t
```

8. Controlling Loops with break and continue

break Statement (Exit Loop Early)

Stops the loop immediately when a condition is met.

```
for (let i = 1; i <= 5; i++) {

    if (i === 3) {

        break; // Stops the loop when i = 3
```

```
    }
    console.log(i);
}
// Output:
// 1
// 2
```

continue Statement (Skip Current Iteration)

Skips the current iteration and moves to the next.

```
for (let i = 1; i <= 5; i++) {
    if (i === 3) {
        continue; // Skips iteration when i = 3
    }
    console.log(i);
}
// Output:
// 1
// 2
// 4
// 5
```

9. Infinite Loops (Be Careful!)

An infinite loop is a loop that never stops because the condition never becomes false.

Example of an Infinite Loop (BAD CODE! 🚨)

```
while (true) {
    console.log("This will run forever!");
}
```

To avoid this, always make sure the loop condition eventually becomes false.

JavaScript Functions

A function in JavaScript is a self-contained block of code designed to perform a specific task. Functions help in modularizing code, improving reusability, and making debugging easier.

1.Why Use Functions?

Functions are useful because they:

- Encapsulate logic → Keep related code in one place.
- Enhance reusability → Write once, use multiple times.
- Improve readability → Code is easier to understand.
- Enable abstraction → Hide implementation details from users.

Example Without a Function (Code Duplication)

```
console.log("The square of 4 is:", 4 * 4);

console.log("The square of 6 is:", 6 * 6);

console.log("The square of 8 is:", 8 * 8);
```

If we need to calculate squares at multiple places, the same logic is repeated.

Example With a Function (Reusability)

```
function square(num) {

    return num * num;

}
```

```
console.log("The square of 4 is:", square(4));

console.log("The square of 6 is:", square(6));

console.log("The square of 8 is:", square(8));
```

Now, the logic is encapsulated in the function, making it reusable and cleaner.

2.Function Types in JavaScript

1. Function Declaration (Named Function)

A function declaration uses the function keyword followed by a name.

```
function functionName(parameters) {
    // Function body
    return value;
}
```

Example:

```
function multiply(a, b) {
    return a * b;
}

console.log(multiply(3, 4)); // Output: 12
```

2. Function Expression (Anonymous Function)

A function expression assigns a function to a variable. These functions can be anonymous (without a name).

Example:

```
const divide = function (a, b) {
    return a / b;
};
console.log(divide(10, 2)); // Output: 5
```

3. Arrow Function (ES6)

Arrow functions provide a concise syntax and automatically bind this.

```
const functionName = (parameters) => expression;
```

Example:

```
const add = (a, b) => a + b;
console.log(add(5, 7)); // Output: 12
```

4. Immediately Invoked Function Expression (IIFE)

An IIFE is a self-executing function that runs immediately after being defined.

Example:

```
(function () {
```

```
    console.log("IIFE executed!");
})();
```

3. Function Parameters & Arguments

1. Default Parameters (ES6)

Default parameters allow predefined values when arguments are not provided.

Example:

```
function greet(name = "Guest") {
    console.log(`Hello, ${name}!`);
}
greet();       // Output: Hello, Guest!
greet("Alex"); // Output: Hello, Alex!
```

2. Rest Parameters (... Operator)

The rest parameter allows passing multiple arguments as an array.

Example:

```
function sum(...numbers) {
    return numbers.reduce((acc, num) => acc + num, 0);
}
```

```
console.log(sum(2, 4, 6, 8)); // Output: 20
```

4. Function Scope & Closures

1. Function Scope

Functions have local scope, meaning variables declared inside are not accessible outside.

Example:

```
function showMessage() {
    let message = "Hello!";
    console.log(message);
}
showMessage(); // Output: Hello!
console.log(message); // Error: message is not defined
```

2. Closures (Functions Inside Functions)

A closure is a function that remembers the variables from its parent's scope even after the parent function has finished execution.

Example:

```
function outer() {
    let count = 0;
```

```
    return function inner() {
        count++;
        console.log(count);
    };
}
const increment = outer();
increment(); // Output: 1
increment(); // Output: 2
```

5.Callbacks & Higher-Order Functions

1. Callback Functions

A callback function is a function passed as an argument to another function.

Example:

```
function processUserInput(name, callback) {
    console.log(`Processing user: ${name}`);
    callback();
}
function displayMessage() {
```

```
    console.log("User processed successfully!");
}

processUserInput("John", displayMessage);

// Output:


// Processing user: John

// User processed successfully!
```

2. Higher-Order Functions

A higher-order function is a function that takes another function as an argument or returns a function.

Example:

```
function multiplyBy(factor) {
    return function (num) {
        return num * factor;
    };
}

const double = multiplyBy(2);

console.log(double(5)); // Output: 10
```

3🟡 Intermediate (Core Concepts)

🔹 **Arrays & Objects** – Used to store multiple values.

Array: Ordered list of values.

```
let fruits = ["Apple", "Banana", "Mango"];

console.log(fruits[1]); // Banana
```

- **Object:** Collection of key-value pairs.

```
let person = { name: "John", age: 30 };

console.log(person.name); // John
```

🔹 **Array Methods** – Built-in functions for array manipulation.

- `.map()` – Returns a new array by transforming elements.
- `.filter()` – Returns a new array with elements that meet a condition.
- `.reduce()` – Reduces array to a single value.
- `.forEach()` – Loops through array elements.

- **Example:**

```
let numbers = [1, 2, 3, 4, 5];
```

```
let squared = numbers.map(num => num * num);

console.log(squared); // [1, 4, 9, 16, 25]
```

🔹 **String Methods** – Built-in functions for strings.

- `.split()` – Converts string to array
- `.join()` – Converts array to string
- `.replace()` – Replaces part of a string
- `.substring()` – Extracts part of a string
- `.trim()` – Removes whitespace

- **Example:**
- ```
  let text = " Hello World ";

  console.log(text.trim()); // "Hello World"
  ```

🔹 **DOM Manipulation** – Modify HTML elements dynamically.

Example:

```
document.getElementById("myElement").innerHTML =
"Hello!";
```

🔹 **Event Handling** – Respond to user actions like clicks and keypresses.

Example:

```
document.querySelector("button").addEventListener("clic
k", function() {
```
```

```
    alert("Button clicked!");
});
```

🔹 **ES6 Features** – Modern JavaScript improvements.

- `let` & `const`
- Template Literals: `console.log(`Hello, ${name}!`);`
- Destructuring:

```
let [a, b] = [10, 20];

console.log(a); // 10
```

- Spread/Rest Operators:

```
let arr1 = [1, 2, 3];

let arr2 = [...arr1, 4, 5];
```

🔵 **Advanced (Deep Understanding)**

🔹 **Asynchronous JavaScript** – Execute code without blocking execution.

- `setTimeout()`, `setInterval()`, Callbacks

Example:

```
setTimeout(() => console.log("Hello after 2 seconds"),
2000);
```

🔹 **Promises & Async/Await** – Handle asynchronous operations.

Example:

```
function fetchData() {
        return new Promise((resolve) => {
            setTimeout(() => resolve("Data fetched"),
    2000);
    });
    }
    async function getData() {
        let data = await fetchData();
        console.log(data);
    }
    getData();
```

- **Error Handling** – Prevent crashes using `try...catch`.

Example:

```
    try {
        let x = y; // y is not defined
    } catch (error) {
        console.log(error.message); }
```

- **LocalStorage & SessionStorage** – Store user data.

```
localStorage.setItem("name", "John");

console.log(localStorage.getItem("name"));
```

🔹 **Closures & Hoisting** –

- **Closures**: Functions inside functions can access outer variables.
- **Hoisting**: Variables declared with `var` move to the top.

🔹 **Prototype & Inheritance** – Object properties can be inherited.

Example:`funct`

```
ion Person(name) {

        this.name = name;

    }

    Person.prototype.greet = function() {

        console.log("Hello, " + this.name);

    };

    let john = new  Person("John");

    john.greet(); // Hello, John
```

🔹 **Event Loop & Call Stack** – JavaScript handles async tasks using an event loop.

Example:

```
console.log("Start");

setTimeout(() => console.log("Async Task"), 0);

console.log("End");

// Output: Start → End → Async Task
```

TypeScript

🟢 Basics (Fundamentals)

1️⃣ What is TypeScript?

- TypeScript is a **superset of JavaScript** that adds **static typing** for better code organization.
- It helps catch **errors during development** instead of runtime.
- **Why use TypeScript?**
  - Detects errors before running the code
  - Helps with **better code structuring**
  - Supports **modern JavaScript features**
  - Improves **team collaboration** with clear type definitions

2️⃣ Installation & Setup

- Install TypeScript globally:

  ```
  npm install -g typescript
  ```
- Check installation:

  ```
  npm install -g typescript
  ```
-

- Create a **TypeScript configuration file (`tsconfig.json`)**:

```
tsc --init
```

3️⃣ Variables & Data Types

- TypeScript has **static types** like:

```
let username: string = "John";

let age: number = 30;

let isAdmin: boolean = true;

let value: any = "Can be anything"; // Avoid using `any`

let user: { name: string; age: number } = { name: "John",
age: 30 };
```

4️⃣ Type Inference & Explicit Types

- **TypeScript can infer types automatically**:

```
let age = 25; // TypeScript understands `age` is a
number
```

- **Or define types explicitly**:

```
let age: number = 25; // Better for clarity
```

5️⃣ Functions & Return Types

- Define function types with parameters and return values:

```
function add(a: number, b: number): number {

return a + b;
```

```
}
```

- **Optional & Default Parameters**:

```
function greet(name: string = "Guest"): string {

return `Hello, ${name}`;

}
```

6️⃣ Type Aliases

- **Reusable types** for objects:

```
type User = { name: string; age: number };

let user: User = { name: "Alice", age: 22 };
```

7️⃣ Interfaces

- **Defines object structure**:

```
interface Person {

name: string;

age: number;

}

let person: Person = { name: "John", age: 30 };
```

🟡 Intermediate (Core Concepts)

8️⃣ Union & Intersection Types

Union Types: Allow multiple possible types

```
let value: string | number;

value = "Hello"; // Valid

value = 42; // Also valid
```

- **Intersection Types:** Combine multiple type

  ```
  type Employee = { name: string; role: string };

  type Manager = { name: string; department: string };

  type TeamLead = Employee & Manager; // Must have
  properties of both
  ```

9️⃣ Tuple & Enum

- **Tuple:** Fixed-length array with specific types

  ```
  let person: [string, number] = ["John", 25];
  ```

- **Enum:** Assigns readable names to values

  ```
  enum Role { Admin, User, Guest }

  let userRole: Role = Role.User;
  ```

🔟 Array & Object Types

- **Typed arrays**:

```
let numbers: number[] = [1, 2, 3];
```

1️⃣1️⃣ Optional & Readonly Properties

- **Optional properties (?)**:

```
type User = { name: string; age?: number }; // `age` is optional
```

- **Readonly properties**:

```
type User = { readonly id: number; name: string };

let user: User = { id: 1, name: "Alice" };

user.id = 2; // ❌ Error: Cannot modify readonly property
```

1️⃣2️⃣ Type Assertions & Casting

- **Convert one type to another**:

```
let input = document.querySelector("#username") as HTMLInputElement;
```

1️⃣3️⃣ Type Guards

```
function printId(id: string | number) {

    if (typeof id === "string") {
```

```
        console.log(id.toUpperCase()); // TypeScript knows
`id` is a string here

    }

}
```

1️⃣4️⃣ Interfaces vs. Type Aliases

- **Interfaces**: Used for **objects** and can be **extended**
- **Type Aliases**: Can be used for **any type** (union, tuple, etc.)

🔵 Advanced (Deep Understanding)

1️⃣5️⃣ Generics

Make functions reusable for different types:

```
function identity<T>(value: T): T {

    return value;

}

let result = identity<number>(5);
```

1️⃣6️⃣ Utility Types

- **Built-in types to modify existing types**:
  `type Person = { name: string; age: number };`

```
type PartialPerson = Partial<Person>; // Makes all
properties optional
```

- `Readonly<T>` – Makes properties readonly
- `Pick<T, K>` – Selects certain keys from a type
- `Omit<T, K>` – Excludes certain keys

1️⃣7️⃣ Mapped Types

- **Dynamically create new types from existing ones**:

```
type Optional<T> = { [K in key of T]?: T[K] };

type PartialUser = Optional<{ name: string; age: number
}>;
```

1️⃣8️⃣ Keyof & Lookup Types

- **Access object keys dynamically**:

```
type User = { name: string; age: number };

type UserKeys = key of User; // "name" | "age"
```

1️⃣9️⃣ Type Narrowing

- **Use control flow to ensure type safety**:

```
function process(value: string | number) {

 if (typeof value === "string") {
```

```
        console.log(value.toUpperCase()); // TypeScript
knows it's a string
    }
}
```

2️⃣0️⃣ Declaration Files (.d.ts)

- **Use TypeScript with third-party libraries**
- Create a `.d.ts` file to define types

2️⃣1️⃣ Module System & Namespaces

- **Use `import/export` to organize large projects**

```
export function greet() { return "Hello!"; }
import { greet } from "./module";
```

2️⃣2️⃣ Strict Mode & Compiler Options

- **Enable strict mode for better type safety**:

```
"strict": true,
 "noImplicitAny": true,
```

- `"strictNullChecks": true`

React

Prerequisites to learn React: HTML, CSS and JavaScript

What is React?..

- React is a JavaScript library for building user interfaces created by Facebook.
- React is used to build single-page applications.
- React allows us to create reusable UI components.
- React uses a virtual DOM to optimize updates, making applications faster and more efficient.

- React uses JSX which combines HTML-like syntax with JavaScript's functionality, making it easy to shift to React from JavaScript.
- Ensures better application control with one-way data binding.

To Start with React you need to install it in your project. Follow these articles to install depending on your system

- How to Install ReactJS on Windows
- How to Install ReactJS on MacOS

Or else follow the below steps

Step 1: Install Node.js
Step 2: Open your terminal in the directory you would like to create your application
Step 3: Run this command to create a React application:

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

Note:

- The app name must be in lowercase due to npm naming restrictions.
- This command installs all required dependencies and sets up your app.

Step 4: Open the Project in a Code Editor
Step 5: Open Terminal and run the following command

```
cd reactapp (your app name)
```

Step 6: Run the React App
Start the development server by running:

```
npm start
```

Congratulations! You have successfully installed the react-app and are ready to build awesome websites and apps.

How does React work?

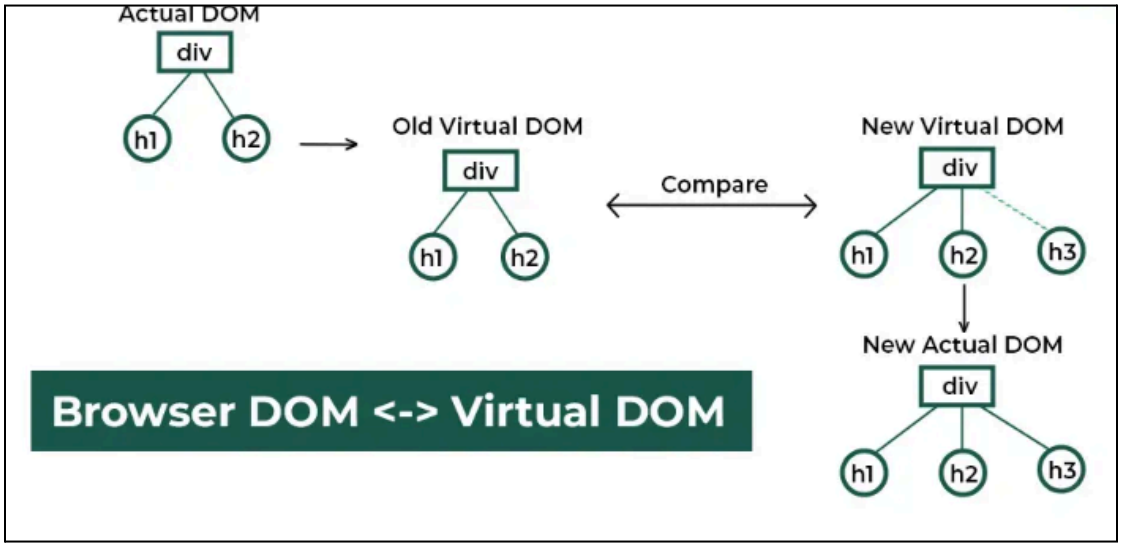
1. React creates a VIRTUAL DOM in memory.

2. Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.

3. React only changes what needs to be changed!

4. React finds out what changes have been made, and changes only what needs to be changed.

Here’s how the process works



1. Actual DOM and Virtual DOM
   - Initially, there is an Actual DOM(Real DOM) containing a div with two child elements: h1 and h2.
   - React maintains a previous Virtual DOM to track the UI state before any updates.

2. Detecting Changes

- When a change occurs (e.g., adding a new h3 element), React generates a New Virtual DOM.
- React compares the previous Virtual DOM with the New Virtual DOM.

3. Efficient DOM Update
   - React identifies the differences (in this case, the new h3 element).
   - Instead of updating the entire DOM, React updates only the changed part in the New Actual DOM, making the update process more efficient.

Features of React

1. Virtual DOM
React uses a Virtual DOM to optimize UI rendering. Instead of updating the entire real DOM directly, React:
- Creates a lightweight copy of the DOM (Virtual DOM).
- Compare it with the previous version to detect changes (diffing).
- Updates only the changed parts in the actual DOM (reconciliation), improving performance.

2.
React follows a component-based approach, where the UI is broken down into reusable components.

3. JSX (JavaScript XML)
React uses JSX, a syntax extension that allows developers to write HTML inside JavaScript.

4. One-Way Data Binding

React uses one-way data binding, meaning data flows in a single direction from parent components to child components via props. This provides better control over data and helps maintain predictable behavior.

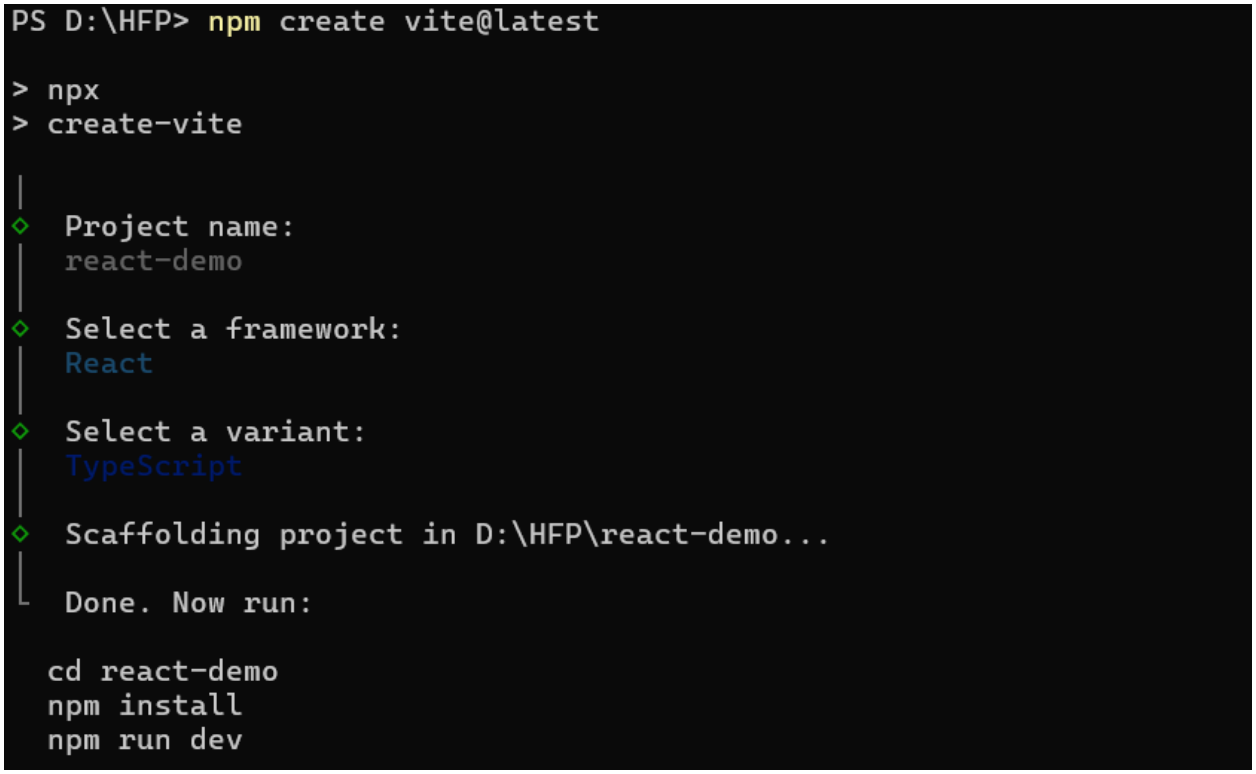
5. State Management
React manages component state efficiently using the useState hook (for functional components) or this.state (for class components). State allows dynamic updates without reloading the page.

6. React Hooks
Hooks allow functional components to use state and lifecycle features without needing class components.

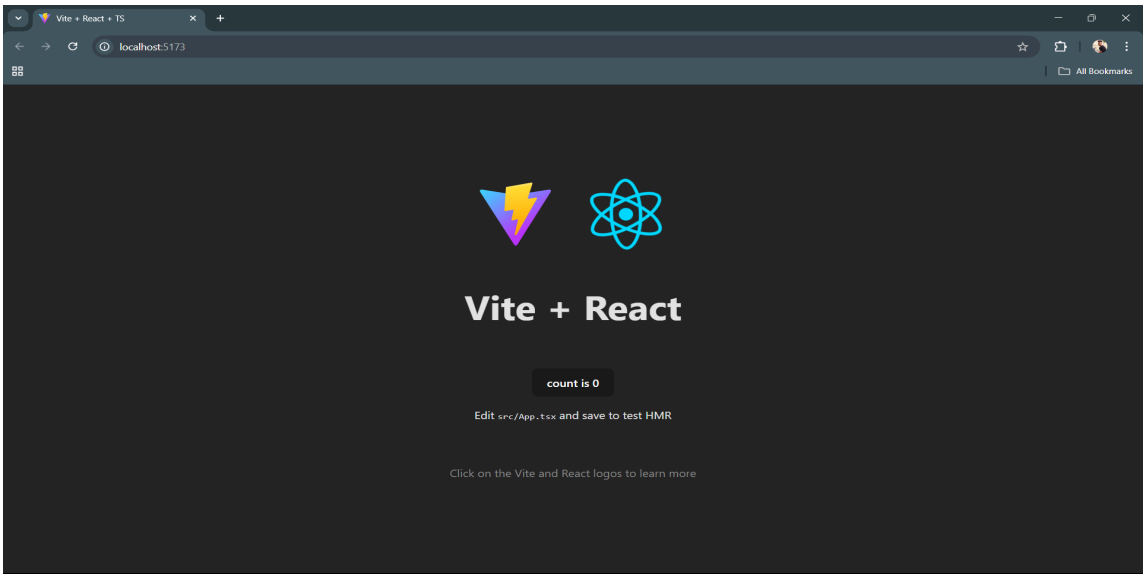
7. React Router
React provides a React Router for managing navigation in single-page applications (SPAs). It enables dynamic routing without requiring full-page reloads.

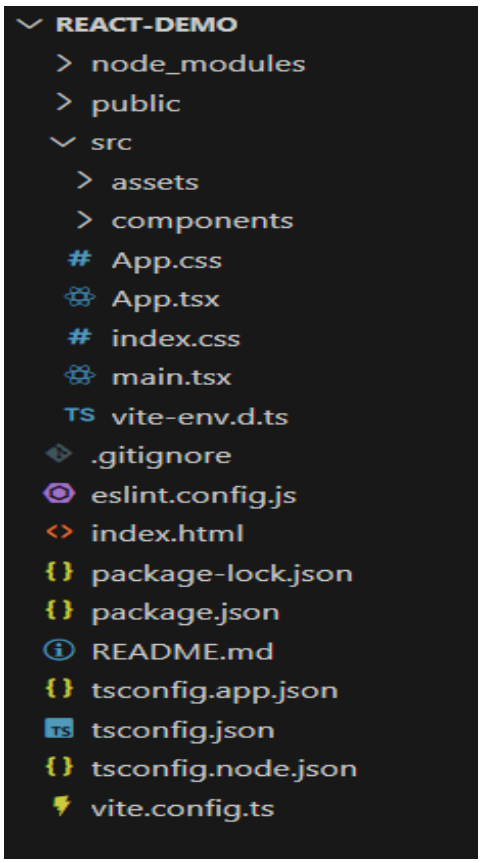
Creating React Application using Vite build tool:

Run following commands on Command Prompt at right file directory



See how your React App looks like





A default application will be created with the following project structure and dependencies

React JSX:

JSX Basics

- JSX stands for JavaScript XML.
- JSX allows us to write HTML in React.
- JSX makes it easier to write and add HTML in React.
- JSX combines HTML and JavaScript in a single syntax, allowing you to create UI components in React and place them in the DOM without any createElement() and/or appendChild() methods.
- JSX converts HTML tags into react elements.

Syntax:

```
const element = <h1>Hello, world!</h1>;
```

- <h1>Hello, world!</h1> is a JSX element, similar to HTML, that represents a heading tag.
- JSX is converted into JavaScript behind the scenes, where React uses React.createElement() to turn the JSX code into actual HTML elements that the browser can understand.

How JSX Works:

When React processes the JSX code, it converts it into JavaScript using Babel. This JavaScript code then creates real HTML elements in the browser's DOM, which is how your web page gets displayed.

Writing JSX:

```
const element = <h1>Hello, World!</h1>;
```

- JSX Gets Transformed: JSX is not directly understood by browsers. So, it gets converted into JavaScript by a tool called Babel. After conversion, the JSX becomes equivalent to React.createElement() calls.
- After transformation JSX becomes:

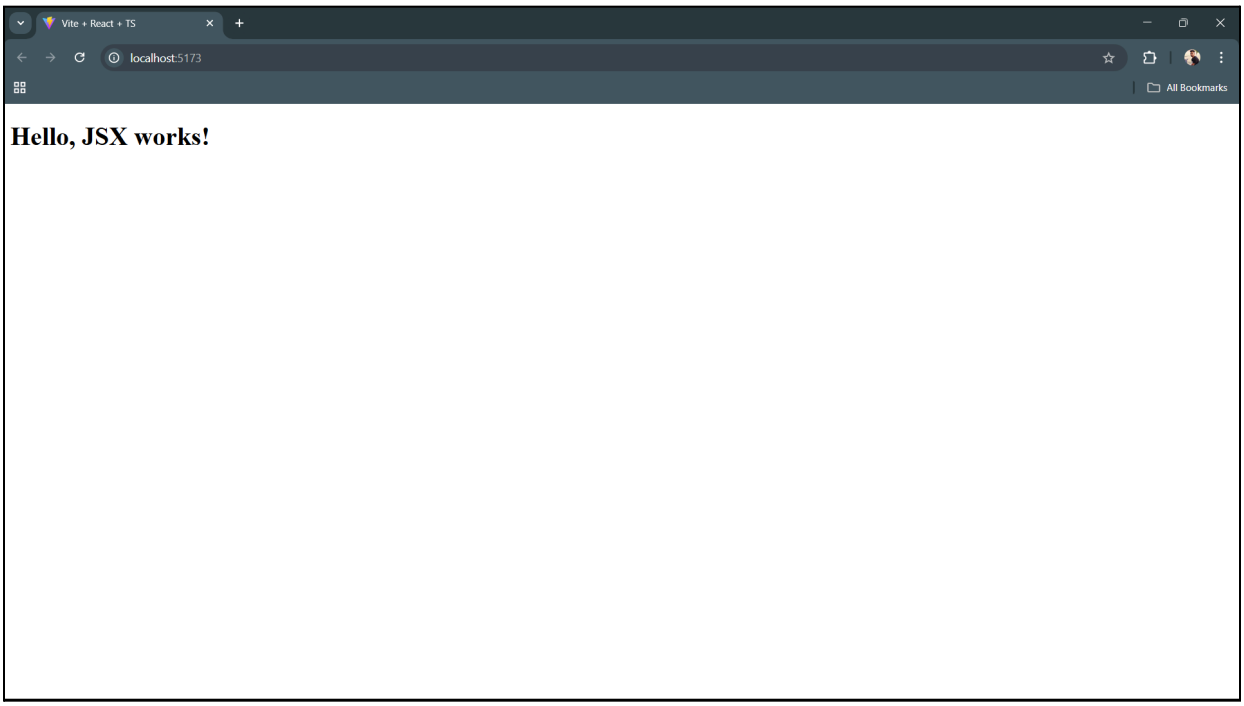
```
const element = React.createElement('h1', null, 'Hello, World!');
```

- React Creates Elements: React takes the JavaScript code generated from JSX and uses it to create real DOM elements that the browser can render on the screen.

Implementation of JSX:

Write JSX in the Component: In the src/App.tsx file

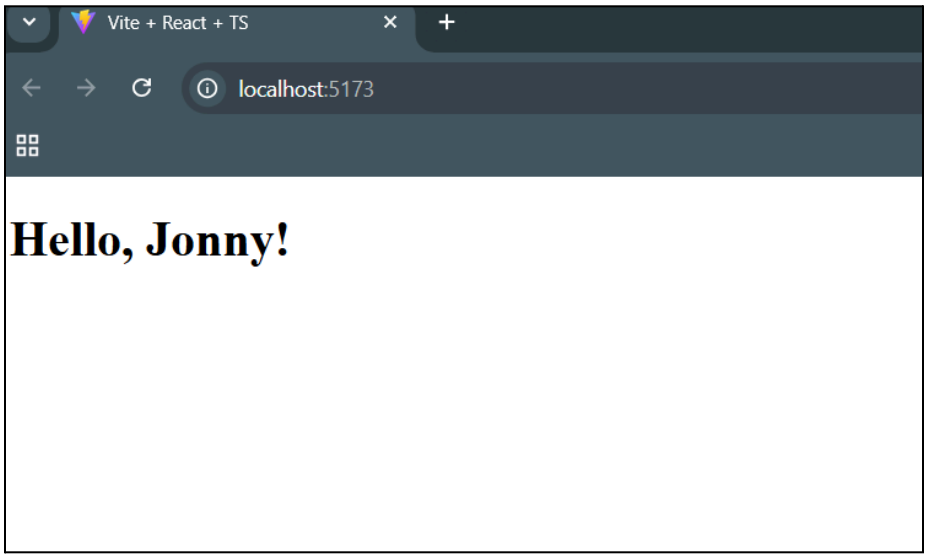
Output:



Some IMP concepts of JSX:

1. Embedding Expressions

- JSX allows you to embed JavaScript expressions directly within the HTML-like syntax. You can use curly braces {} to insert JavaScript expressions.
- The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the output.



2. Inserting large block of HTML

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

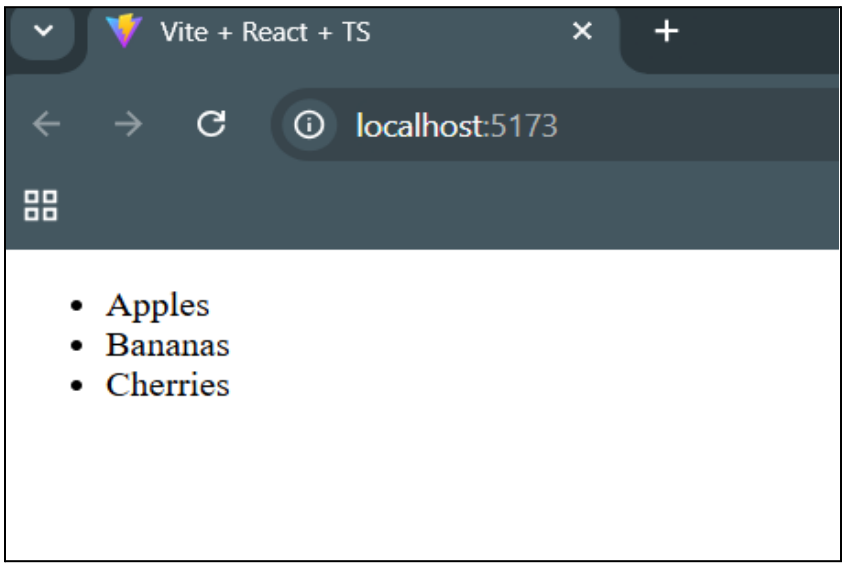
  Creating a list with three list items:

```tsx
// App.tsx
import React from "react";

function App() {
  const myElement = (
    <ul>
      <li>Apples</li>
      <li>Bananas</li>
      <li>Cherries</li>
    </ul>
  );

  return myElement;
}

export default App;
```



3. One Top Level Element

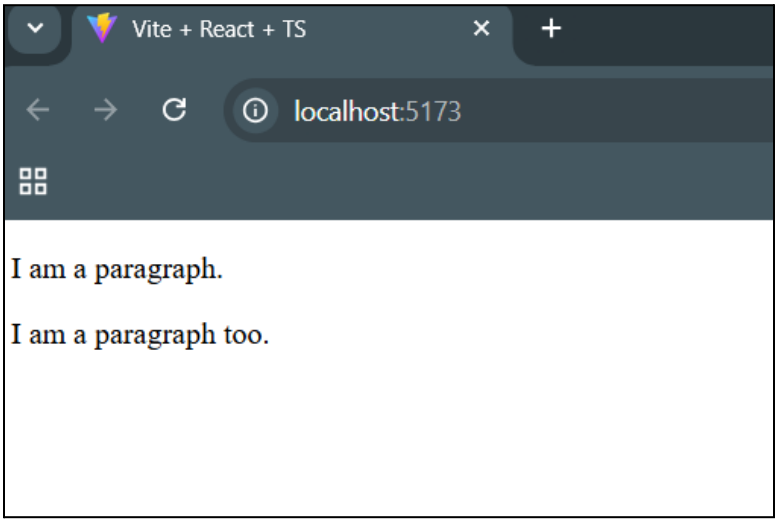
- The HTML code must be wrapped in ONE top level element.
- So if you like to write two paragraphs, you must put them inside a parent element, like a div element.

Wrapping two paragraphs inside one DIV element:

App.tsx 1

src > App.tsx > ...

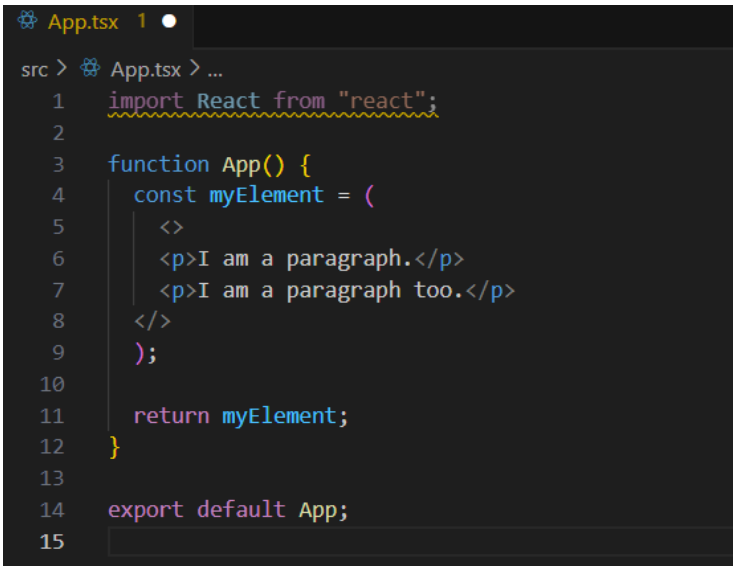
```
1   import React from "react";
2
3   function App() {
4     const myElement = (
5       <div>
6         <p>I am a paragraph.</p>
7         <p>I am a paragraph too.</p>
8       </div>
9     );
10
11    return myElement;
12  }
13
14  export default App;
15
```



Note:
JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.

- we can also use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.

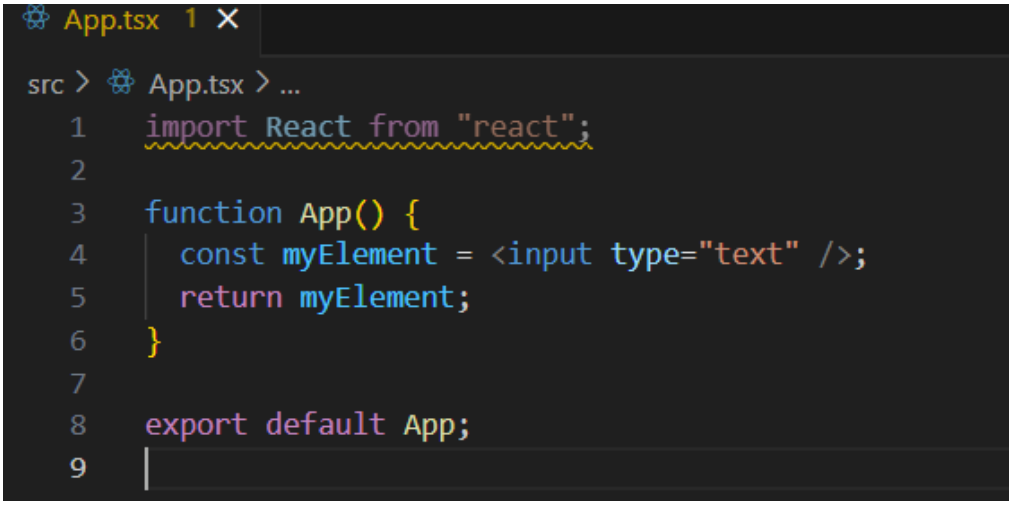
- A fragment looks like an empty HTML tag: <></>.

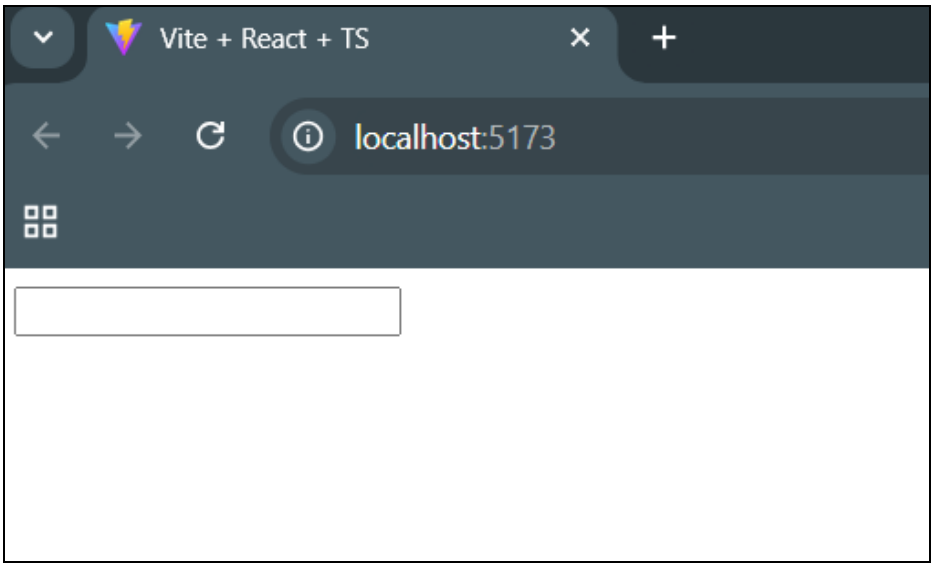


4. Elements must be closed

- JSX follows XML rules, and therefore HTML elements must be properly closed.

Close empty elements with />





Note:
JSX will throw an error if the HTML is not properly closed.

5. Attribute class = className

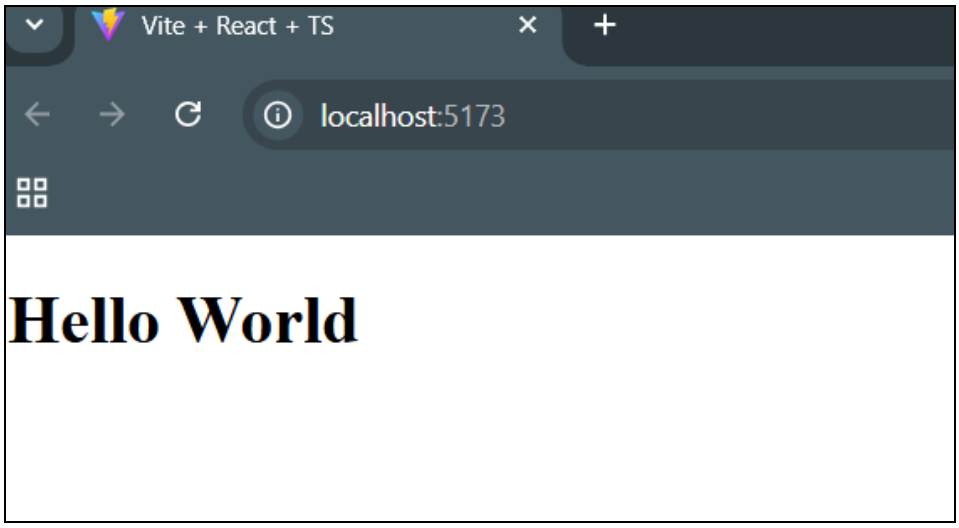
- The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.
- Use attribute className instead.
- JSX solved this by using className instead. When JSX is rendered, it translates className attributes into class attributes.

Use attribute className instead of class in JSX:

App.tsx 1

src > App.tsx > ...

```
1  import React from "react";
2
3  function App() {
4    const myElement = <h1 className="myclass">Hello World</h1>;
5    return myElement;
6  }
7
8  export default App;
9
```



React Babel:

- Open-source JavaScript compiler and transpiler.
- Enables the use of modern JavaScript (ES6+) features.
- Ensures compatibility across different browsers.
- Helps React developers write clean, modern syntax.

How Babel Works with ReactJS:

1. React uses JSX, which browsers don't natively understand.
2. Babel converts JSX into standard JavaScript functions.
3. Uses the @babel/preset-react plugin for JSX transformation.

React Lists:

- React Lists are used to display collections of similar data items, such as arrays of objects or menu items.
- They enable dynamic rendering of array elements to display repetitive data efficiently.

Rendering List in React

- In React, lists are used to display multiple similar data items dynamically.
- The JavaScript map() function is commonly used to iterate over an array and render elements in JSX.

App.tsx 2 ● FruitsList.tsx 1 ×

src > components > FruitsList.tsx > FruitsList > items

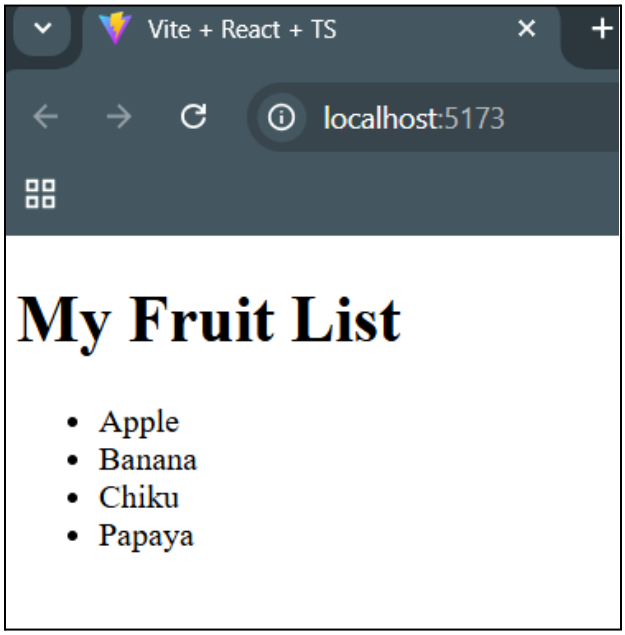
```jsx
import React from "react";

const FruitsList = () => {
  const items = ["Apple", "Banana", "Chiku", "Papaya"];
  return (
    <div>
      <h1>My Fruit List</h1>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
};

export default FruitsList;

```

Output:



Explanation

- The .map() function iterates over the items array.
- Each item is rendered inside an <li> tag.
- The key prop is used to uniquely identify each list item.

React Forms:

- Forms in React are used to collect and manage user input efficiently. React uses controlled components to handle form data within the component state instead of relying on the traditional DOM.
- Forms collect user input for various purposes like authentication, payment processing, and data collection.
- React forms store data in component state, making them easy to manage.
- Unlike traditional forms, React prevents default behavior (like page refresh) when submitting forms.

Syntax:

```
<form action={handleSubmit}>
  <label>
     User name:
     <input type="text" name="username" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Adding Forms in React

- Forms in React can be easily added like a regular HTML form but with additional React functionalities, such as state management and event handling.
- React uses the onChange event to track user input and update values dynamically.

This example captures user input and logs it to the console when the onChange event is triggered.

App.tsx 6 ReactForms.tsx

src > components > ReactForms.tsx > ReactForms

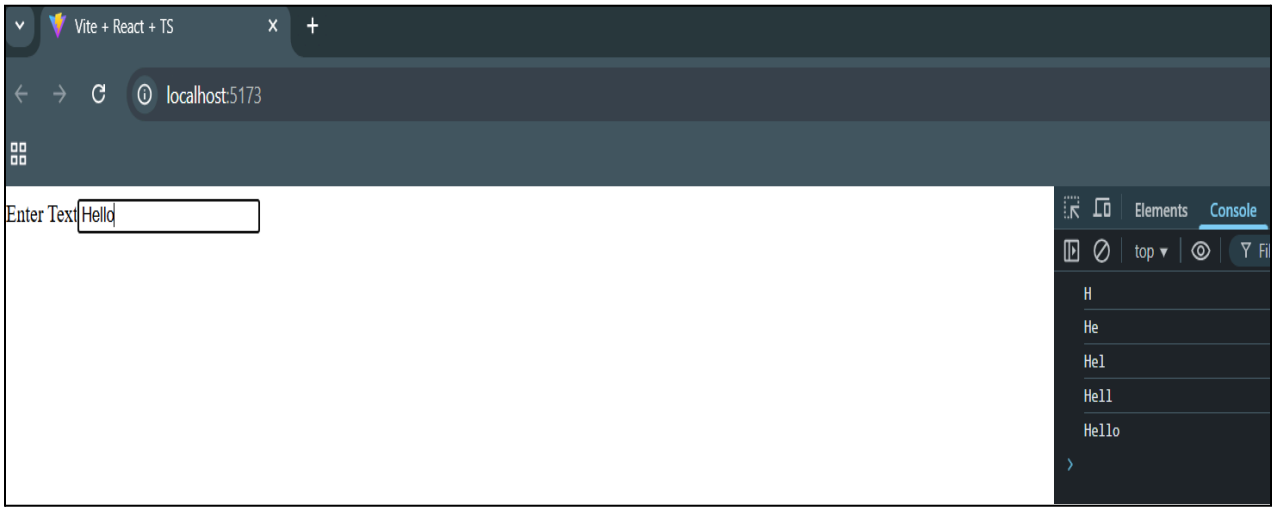
```tsx
import React from "react";

const ReactForms = () => {
  const onInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    console.log(e.target.value);
  };

  return (
    <div>
      <form>
        <label>Enter Text</label>
        <input type="text" onChange={onInputChange} />
      </form>
    </div>
  );
};

export default ReactForms;
```

Output:



Explanation:

- onInputChange is an event handler function that gets triggered whenever the user types in the input field.
- e represents the event object of type React.ChangeEvent<HTMLInputElement>, ensuring type safety in TypeScript.
- e.target.value retrieves the current value entered by the user in the input field.
- console.log(e.target.value); prints the input value to the console.
- The <form> element contains:

- A <label> with the text "Enter Text" to describe the input field.
- An <input> field of type text with an onChange event that calls onInputChange whenever the user types something.

React Keys:

- Keys are special attributes in React used to identify elements in a list uniquely.
- They help React optimize rendering performance by tracking changes efficiently.

Importance of Keys

- Improve rendering performance by allowing React to update only the changed elements.
- Help React identify which items are added, removed, or re-ordered.
- Prevent unnecessary re-renders and improve UI efficiency.

Note:

- Use unique identifiers (e.g., database IDs) instead of array indexes.
- Avoid using random or non-deterministic values as keys.
- If using index as a key, ensure the list items won't be reordered or removed frequently.

React Components:

- Components are the building blocks of React applications.
- They allow UI to be broken into independent, reusable pieces.
- Components function similarly to JavaScript functions but return HTML elements (JSX).

Why Split Components into Separate Files?

- Improves code reusability and maintainability.
- Keeps files clean and organized.
- Makes debugging and collaboration easier.

Creating a Component in a Separate File

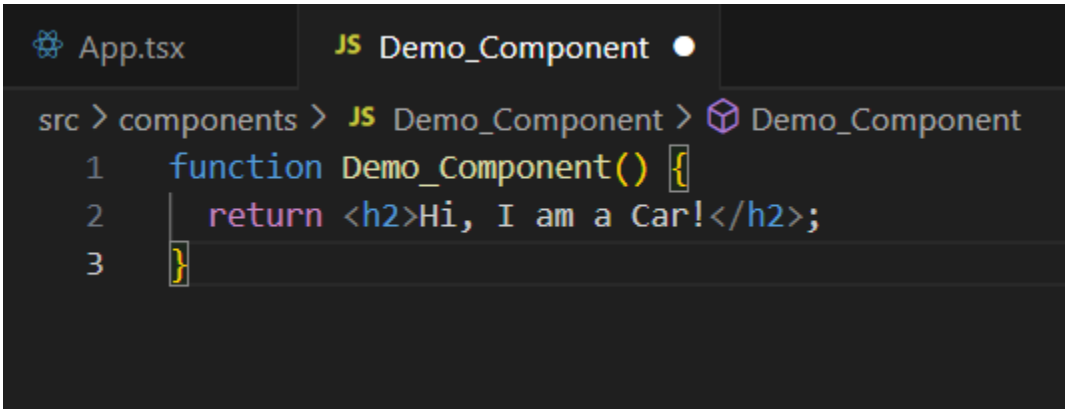
- The component file must have a .tsx extension.
- The filename should start with an uppercase letter (React components follow PascalCase).

Types of React Components

1. Function Components (Preferred with Hooks)
2. Class Components (Older approach, now less commonly used)

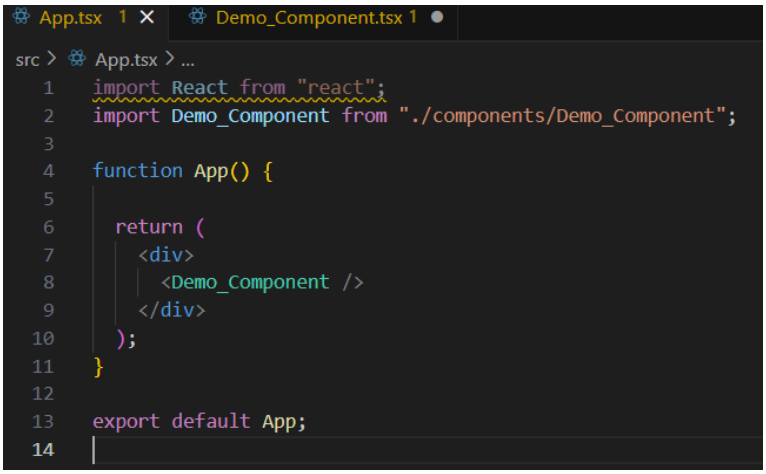
Functional Components:

- Function components are the simpler and modern way to create React components.
- They return JSX (HTML-like syntax in JavaScript).
- Preferred over class components due to conciseness, readability, and better performance



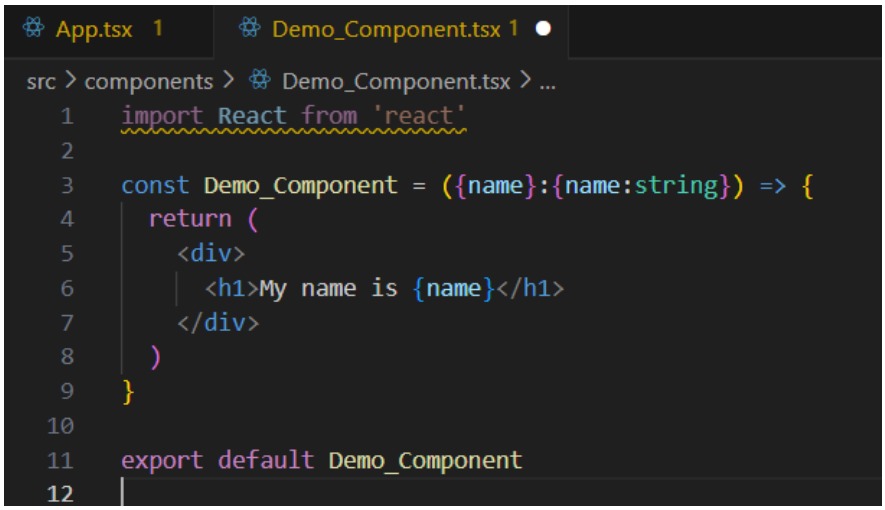
Rendering a Component

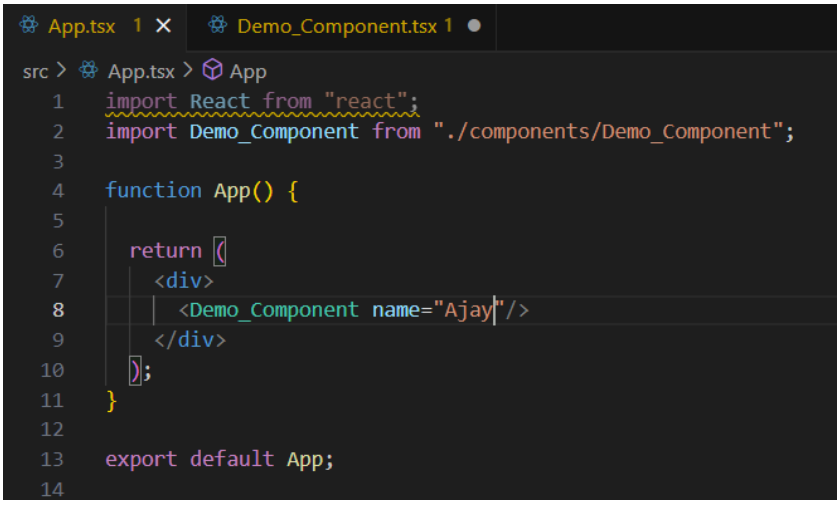
- Components are rendered using <ComponentName /> inside App.tsx.
- To use a component in another file, export and import it.



Props in Components:

- Props (short for properties) allow passing data from a parent component to a child component.
- They work like function arguments and help make components reusable.
- Props are read-only (immutable) and cannot be modified inside the component.



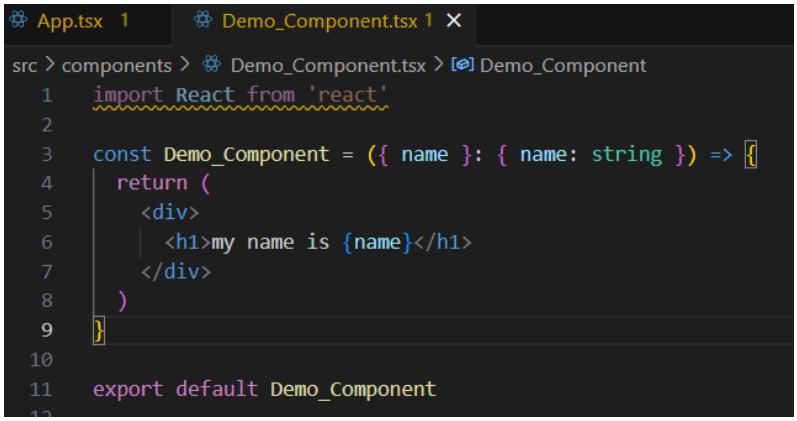


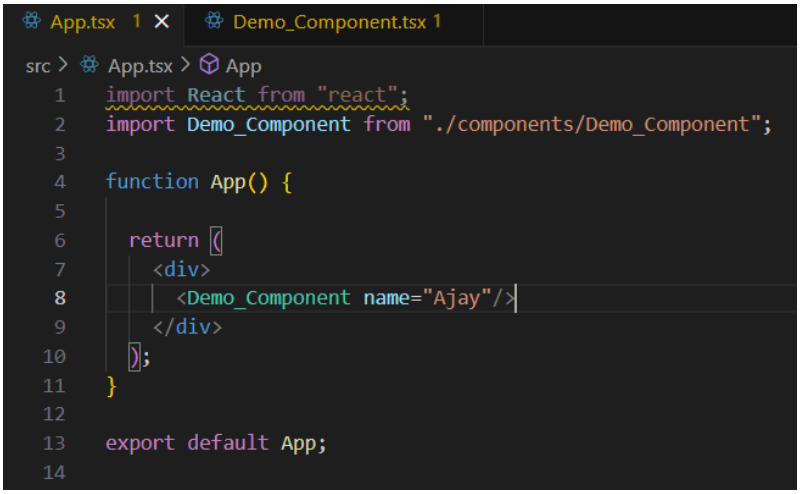
React Props:

- Props (short for properties) are arguments passed into React components.
- They allow data to be passed from a parent component to a child component.
- Props make components reusable and dynamic by allowing customization.
- Props cannot be modified inside the component.
- They are immutable and should be used as they are passed.

Passing Props:

- Props are passed to components via HTML-like attributes.





PropTypes:

- PropTypes is used to enforce type safety in React props.
- It ensures that components receive props of the correct type

Benefits of PropTypes:

- Prevents runtime errors due to incorrect data types.
- Helps with debugging by providing warnings in the console.
- Acts as self-documentation for components.

```
const Demo_Component = ({ name }: { name: string }) => {
  return (
    <div>
      <h1>my name is {name}</h1>
    </div>
  )
}
```

React State:

- React State is an object that stores dynamic data for a component and determines its behavior.
- When the state updates, React automatically re-renders the component to reflect the changes.

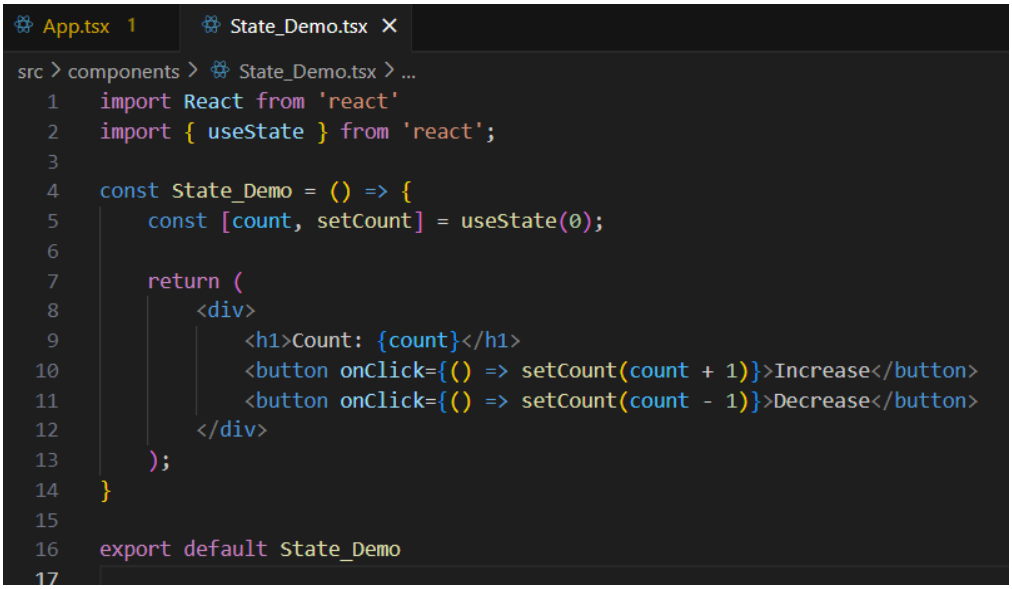
- State makes components interactive and dynamic.

Creating State in React:

- In class components, state is initialized inside the constructor using this.state = {}.
- In functional components, state is managed using the useState hook.

Conventions for Using State in React:

- State should be initialized in the constructor for class components.
- State should not be modified directly; always use setState() to update it.
- State updates may be batched by React for performance optimization.
- Use functional updates (prevState) when updating based on the previous state.

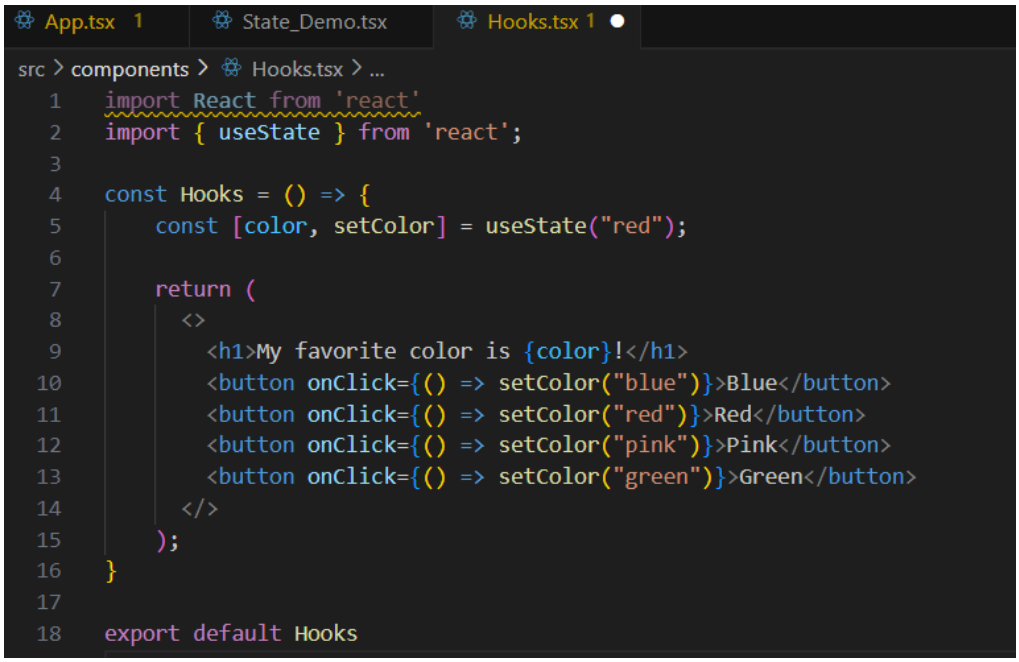


React Hooks:

- Hooks were introduced in React 16.8 to allow function components to access state and other React features.
- Hooks eliminate the need for class components, making the code cleaner and more maintainable.
- React has no plans to remove class components; both approaches can coexist.

What is a Hook?

- Hooks enable function components to "hook into" React features like state and lifecycle methods.



1. Hooks can only be called inside React function components.
2. Hooks must be called at the top level of a component.
3. Hooks cannot be used conditionally.

Types of React Hooks

1. State Hooks
   useState allows function components to manage local state.
2. Context Hooks
   useContext enables access to React Context API for state sharing across components.

3. Ref Hooks
   useRef allows access to DOM elements without re-rendering.
4. Effect Hooks
   useEffect handles side effects like data fetching and event subscriptions.

Benefits of Hooks:

- Simpler Code: Functional components with hooks are concise and easy to understand.
- Reusable Logic: Custom hooks enable easy logic reuse without altering component structure.
- Better State Management: useState and useReducer simplify state handling.
- Improved Maintainability: Code is easier to test and debug.

Why React Introduced Hooks?:

- Avoid 'this' Keyword Confusion: Class components require careful handling of this.
- Reusable Stateful Logic: Hooks eliminate the complexity of Higher-Order Components (HOCs) and Render Props.
- Simplified Complex Logic: Hooks allow grouping related logic in a single function.

Rules for Using Hooks:

- Only functional components can use hooks.
- Hooks must be imported from React.
- Hooks should be called at the top level of components.
- Hooks should not be inside conditional statements.

React Router:

- React Router is a library used for handling navigation in React applications.
- It enables dynamic routing, allowing the UI to change based on the URL without refreshing the page.

- Essential for Single Page Applications (SPAs), where only specific components update instead of the whole page.

Features of React Router:

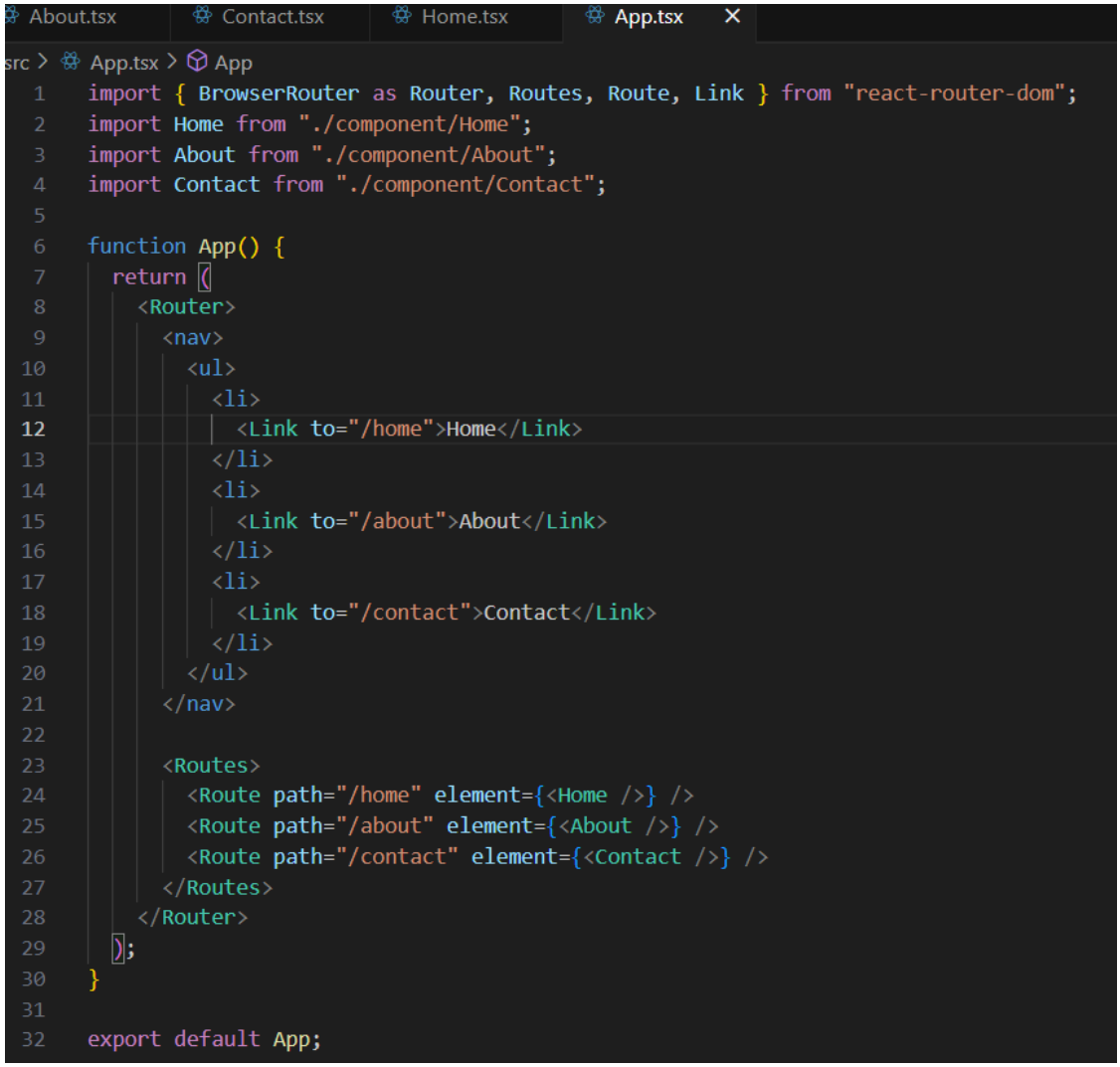
- Declarative Routing: Uses <Routes> and <Route> to define routes in a readable and structured way.
- Nested Routes: Allows creating a hierarchy of routes for better organization.
- Programmatic Navigation: The useNavigate hook enables navigation based on conditions.
- Route Parameters: Supports dynamic URL parameters for flexible routing.
- Improved TypeScript Support: Ensures type safety for better development.

Components of React Router:

1. BrowserRouter & HashRouter:
    a. <BrowserRouter>: Uses the browser's history API for clean URLs.
    b. <HashRouter>: Uses the hash (#) in the URL to track navigation.
2. Routes & Route:
    a. <Routes>: Container that holds multiple <Route> components.
    b. <Route>: Defines the path and component to render.
3. Link & NavLink:
    a. <Link>: Used for internal navigation (instead of <a href="">).
    b. <NavLink>: Similar to <Link>, but provides active styling when selected.
4. Outlet:
   Used in parent routes to display child route components dynamically.

To install React Router in project:

npm install react-router-dom



Uses of React Router:

- Navigation & Routing: Enables smooth navigation between different views in an app.
- Dynamic Routing: Routes can change dynamically based on state or user actions.
- URL Management: Supports deep linking, bookmarking, and history tracking.
- Component-Based Routing: Routes are modular and reusable, following React's component-based approach.

React Fragment:

- React Fragments allow grouping multiple elements without adding an extra node to the DOM.
- They help return multiple child elements from a component without a parent container like .

Why Use React Fragments?

1. Cleaner DOM Structure: Avoids redundant wrapper elements.
2. Better Performance: Reduces extra nodes, improving rendering speed.
3. Avoids Unnecessary Markup: Prevents excessive elements.
4. Flexibility: Allows returning multiple elements while maintaining React's single parent requirement.

Syntax:

```
<>
        <h1>Hello, World!</h1>
        <p>Welcome to the world of React!</p>
</>
```

Ways to Use React Fragments

1. Using Shorthand (<> and </>)

```
return (
  <>
    <h1>Title</h1>
    <p>Content</p>
  </>
```

);

- The simplest and most common way to use fragments.
- Uses empty tags to wrap multiple elements.

2. Using React.Fragment

```
<React.Fragment>
    <h1>Title</h1>
    <p>Content</p>
</React.Fragment>
```

- Allows using the key prop, which is useful for lists.
- More detailed but offers extra functionality.

JavaScript

JavaScript
TypeScript
Redux …React
Next.js
TailwindCSS

Tailwind CSS

Topics given by yash to add

Next.js Fundamentals

React Basics (JSX, components, props, state, hooks)

Next.js File-based Routing (pages & API routes)

Server-side Rendering (SSR) & Static Site Generation (SSG)

Client-side Data Fetching (useEffect, SWR, useSWR)

API Routes for Backend Integration

Middleware & Authentication (e.g., NextAuth.js)

2. Tailwind CSS

Utility-first styling approach

Responsive design classes (sm, md, lg, xl, 2xl)

Customizing themes using tailwind.config.js

Flexbox & Grid Layouts

Dark mode support

3. Additional Concepts

State Management (Context API, Redux, Zustand)

Fetching APIs using Axios or Fetch

SEO & Performance Optimization

Deployment (Vercel, Netlify, or AWS)

Form Handling (Formik, React Hook Form)