**TypeScript**

TypeScript is a superset of JavaScript.

It adds static typing to JavaScript — meaning you can declare variable types and catch errors at compile time (before code runs).

It compiles to plain JavaScript that browsers can run.

**Installation**: go to terminal and run “npm i -g typescript” to globally install TS in your system.

**Importance**

Type Safety – Helps catch errors early (during development) I.e JavaScript shows errors only at runtime, whereas TypeScript catches errors while writing code using static type checking.

Better IDE Support – Code suggestions, auto-completion, refactoring become much smarter.

Scalability – Makes large codebases easier to manage and maintain.

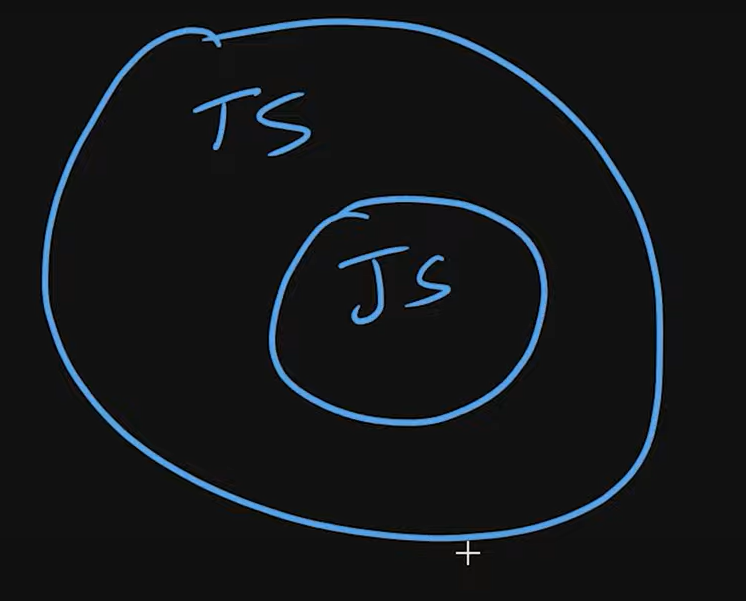
OOP Features – Offers interfaces, generics, enums, and classes — supports additional features of object-oriented programming like access modifiers.

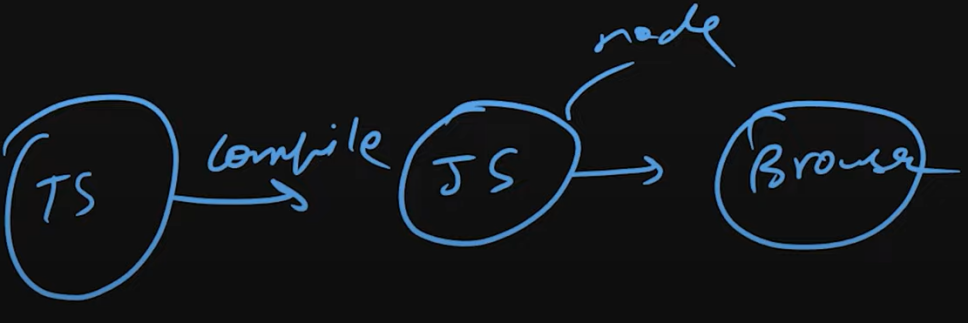
Industry Standard – Used in projects like Angular, Deno, VSCode, and large-scale React apps.

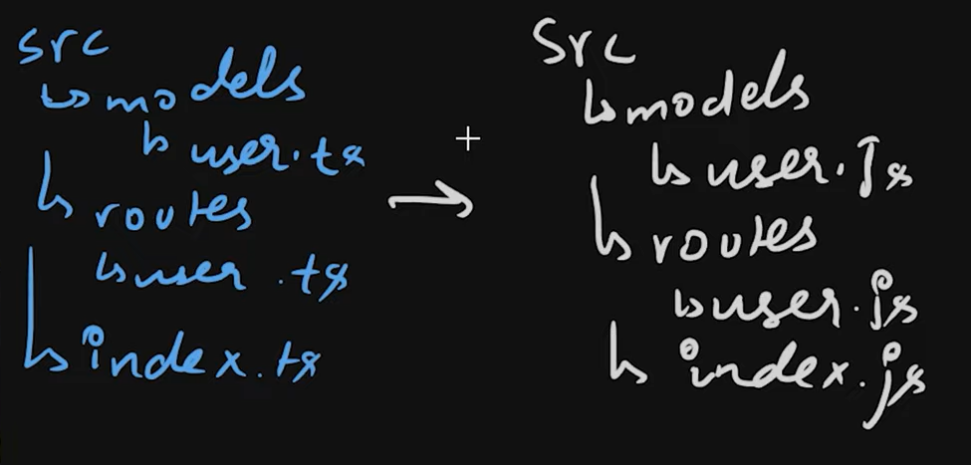
**TypeScript vs JavaScript**

| Feature | JavaScript | TypeScript |
| --- | --- | --- |
| Typing | Dynamic | Static (optional) |
| Error Checking | Runtime | Compile-time |
| Compilation | No (interpreted) | Yes (compiles to JS) |
| IDE Support | Basic | Advanced (intellisense) |
| Usage | Frontend + Backend | Frontend + Backend |

****

****

****

****

**Why can't browsers understand React, TypeScript, or JSX?**

Browsers can only understand HTML, CSS, and vanilla JavaScript.

They do not understand:

* JSX (used in React) → const element = <h1>Hello</h1>
* TypeScript (has types, interfaces)
* SCSS/SASS or other preprocessor code

So, we use build tools (like Vite, Webpack, Parcel) to convert these into browser-readable files.

**✅ What happens when you run npm run build?**

It starts the build process using a tool like **Vite**/Webpack (depending on your setup).

The tool:

* Compiles JSX → JavaScript
* Compiles TS → JS
* Bundles all JS files together (for better performance)
* Minifies the code (removes whitespace, comments, shortens names)
* Places the final HTML, CSS, JS into a dist/ or build/ folder

These final files are what the browser receives and executes.

**✅ Real-World Analogy**

Think of React/TS/JSX as raw ingredients.

npm run build is the cooking process that turns them into a final dish (HTML, CSS, JS) the browser can “eat.”

**✅ What happens during development (npm run dev)?**

Uses a development server (like Vite or Webpack Dev Server)

It does live compilation — meaning:

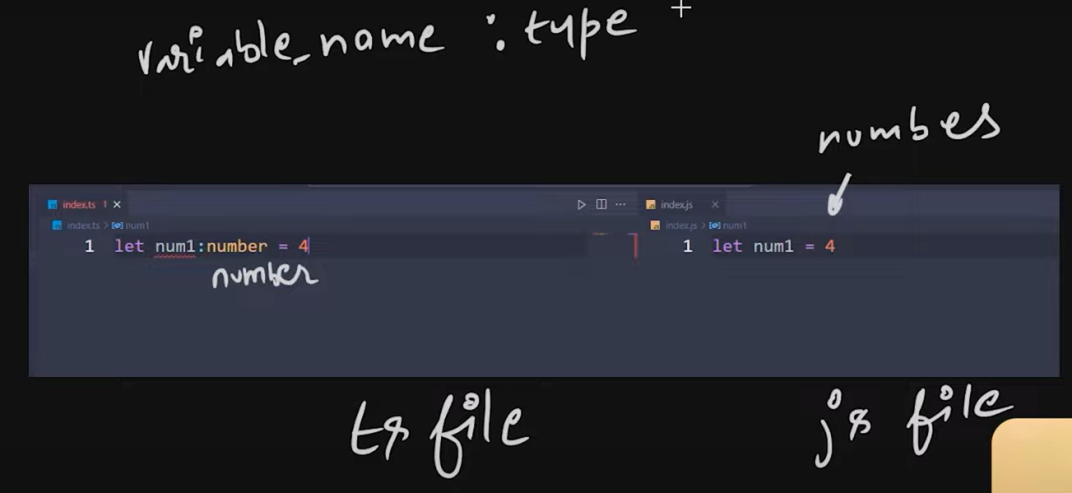
When you write JSX/TS, it compiles it instantly in the background

Browser gets updated without a full refresh (thanks to hot module replacement or HMR)

**✅ Interview-Ready Summary:**

The browser only understands HTML, CSS, and JavaScript. Tools like React, TypeScript, or JSX need to be compiled first. When we run npm run build, the build tool (like Vite or Webpack) converts our modern code into optimized HTML/CSS/JS that browsers can execute.

**TypeScript: Variables and Data Types**



-Variables are declared using let, const, or var, and can have types explicitly defined.

let age: number = 25;

-TypeScript supports basic data types: string, number, boolean, null, undefined, any, void, never.

any type disables type checking, allowing any value.

let data: any = "text";

unknown is safer than any because it forces type checking before use.

-Use const for variables that won't change, let for variables that can.

-Type inference allows TypeScript to guess the type if not explicitly mentioned.

let name = "Faziel"; // inferred as string

-Union types allow a variable to hold more than one type.

let id: number | string = 101;

-Use typeof to check the runtime type of a variable.

**TypeScript Function Notes**

# 1. Basic Function with Parameter Type

Code Example:  
function hello(name: string) {  
 console.log("Hii", name);  
}  
hello("Nishant");

In TypeScript, we can define parameter types using `parameterName: type`, ensuring the correct data is passed.

# 2. Function with Return Type

Code Example:  
function sum(val1: number, val2: number): number {  
 return val1 + val2;  
}

We can specify the return type of a function using `: type` after the parameter list to ensure it returns the correct data type.  
Example: function sum(a: number, b: number): number returns a number.

# 3. Union Types and Type Checking in Function

Code Example:  
function combine(val1: string | number, val2: string | number) {  
 if (typeof val1 === "number" || typeof val2 === "number") {  
 return val1.toString() + val2.toString();  
 }  
 return val1 + val2;  
}

We can allow multiple types using union types (`|`), and use `typeof` to check types at runtime before operating on values. This is helpful when the function can handle both strings and numbers safely.

# **Arrays in TypeScript**

## 1. Typed Arrays

Syntax: let arr: type[] = [values];

Use: Restricts array to a single type.

Example:

let prices: number[] = [100, 200];  
prices.push(300); // ✅ OK  
prices.push("free"); // ❌ Error

## 2. Union Types in Arrays

Syntax: let arr: (type1 | type2)[] = [values];

Use: Allows multiple types (but still type-checked).

Example:

let data: (string | number)[] = ["Admin", 1];  
data.push("User"); // ✅  
data.push(2); // ✅  
data.push(true); // ❌

## 3. Readonly Arrays

Syntax: readonly type[]

Use: Makes the array immutable (no push/pop/splice).

Example:

let roles: readonly string[] = ["view", "edit"];  
roles.push("delete"); // ❌ Error  
roles[0] = "readonly"; // ❌ Error

## 4. Type Inference & any[]

Inference: TypeScript auto-detects types.

let mixed = [1, "two"]; // inferred as (string | number)[]

any[]: Accepts any type (no type safety).

let stuff: any[] = [1, "text", {}, null];  
stuff.push(true); // ✅ OK

## 5. Suppressing Errors in TypeScript

`// @ts-ignore`: Ignores next line’s error (⚠️ risky).

// @ts-ignore  
arr.push(1); // Error suppressed

`// @ts-expect-error` (✅ Better): Suppresses expected error, alerts if no error.

Example:

// @ts-expect-error  
roles.push("admin");

## 🔁 Quick Summary Table

|  |  |
| --- | --- |
| Syntax | Meaning |
| number[] | Array of numbers only |
| (string | number)[] | Array of strings or numbers |
| readonly string[] | Immutable string array |
| any[] | Accepts any type (⚠️ avoid if possible) |

# Tuples in TypeScript

🔹 Tuple: An array with a predefined length and predefined types for each index.

## 1. Basic Tuples

Tuples have a fixed length and each index has a specific type.  
Syntax: let tupleName: [type1, type2, ...] = [value1, value2, ...];

* ✅ Example:

let rgbColor: [number, number, number];  
rgbColor = [255, 0, 120]; // OK

* ❌ Invalid Examples:

rgbColor = [255, 0]; // Error: Only 2 values given.  
rgbColor = [255, 0, 120, 0.5]; // Error: Too many values.  
rgbColor[0] = "red"; // Error: string not assignable to number.

## 2. Common Pitfall: push() Method Quirk

Tuples can be modified using push() even though they are supposed to be fixed-length.

* Example:

let coordinate: [number, number] = [12323, 12344532];  
coordinate.push(4); // ✅ Allowed (but not good)  
console.log(coordinate); // [12323, 12344532, 4]

⚠️ This breaks the tuple's fixed-length rule and may cause bugs.

## 3. Solution: readonly Tuples

Use 'readonly' to make tuples truly fixed and immutable.

* Example:

let coordinate: readonly [number, number] = [10, 20];  
coordinate.push(30); // ❌ Error  
coordinate[0] = 5; // ❌ Error

✅ readonly prevents reassignment and push/pop.

## 🔑 Key Takeaway

Tuples are fixed-length arrays with types at each index.  
❗ TypeScript allows push() on regular tuples, which is a quirk.  
✅ Best Practice: Always use readonly for safe, immutable tuples.

Example: let user: readonly [number, string] = [1, "Alice"];

## Type Aliases & Interfaces (Defining Object Shapes)

In TypeScript, you can define a reusable 'shape' or structure for your objects. While you can define this shape directly on a variable (inline), the best practice is to use a Type Alias (type) or an interface.

## 1. Defining Object Types

You must define all properties that an object should have.

* Required Properties: If a property is listed in the type definition, it must exist in the object.

```typescript  
let user: {  
 name: string;  
 email: string;  
 age: number;  
} = {  
 name: "Nishant",  
 email: "nishant@gmail.com",  
}; // Error: Property 'age' is missing.  
```

* Optional Properties: Use a `?` after the property name to make it optional.

```typescript  
let user: {  
 name: string;  
 email: string;  
 age?: number; // 'age' is now optional  
} = {  
 name: "Nishant",  
 email: "nishant@gmail.com",  
}; // OK!  
```

* Good to know: An optional property `age?: number` is just a shortcut for `age: number | undefined`.

## 2. The Better Way: Using `type` Aliases

Defining the object shape inline is repetitive. It's much cleaner to create a named Type Alias.

* Syntax: `type TypeName = { ... };`
* Benefit: This makes the type reusable, keeping your code clean and easy to read.

```typescript  
type User = {  
 name: string;  
 email: string;  
 age?: number; // Optional property  
 readonly id: number;  
};  
  
const user1: User = {  
 id: 1,  
 name: "Alice",  
 email: "alice@example.com",  
 age: 28,  
};  
  
const user2: User = {  
 id: 2,  
 name: "Bob",  
 email: "bob@example.com",  
};  
  
// user1.id = 5; // Error: Cannot assign to 'id' because it is a read-only property.  
```

## 3. Arrays of Objects

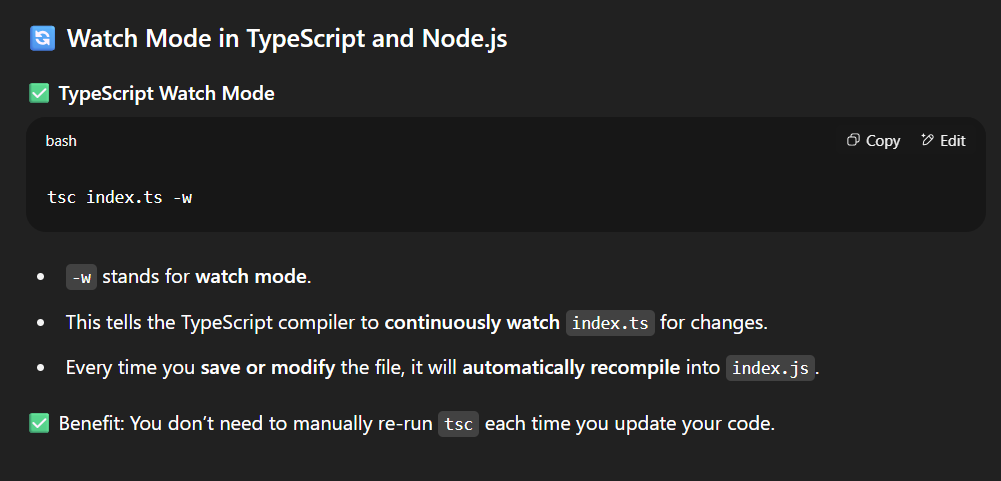
To create an array of objects, simply add `[]` after your type alias name.

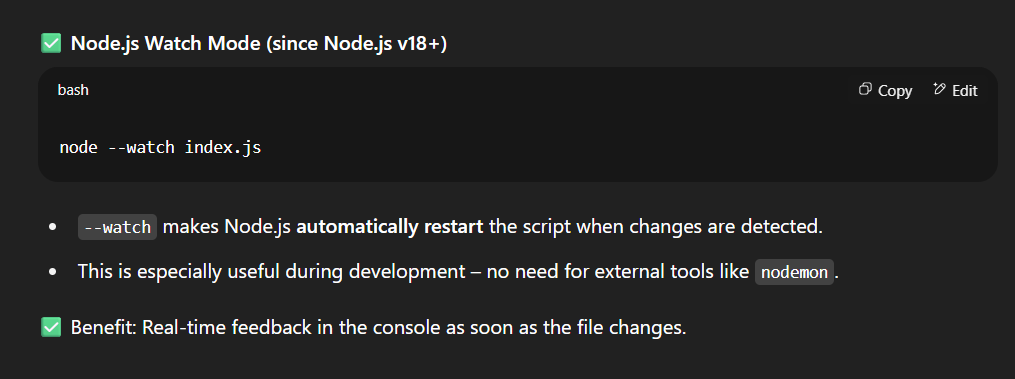
```typescript  
const allUsers: User[] = [user1, user2];  
  
console.log(allUsers);  
// Output:  
// [  
// { id: 1, name: 'Alice', email: 'alice@example.com', age: 28 },  
// { id: 2, name: 'Bob', email: 'bob@example.com' }  
// ]  
```

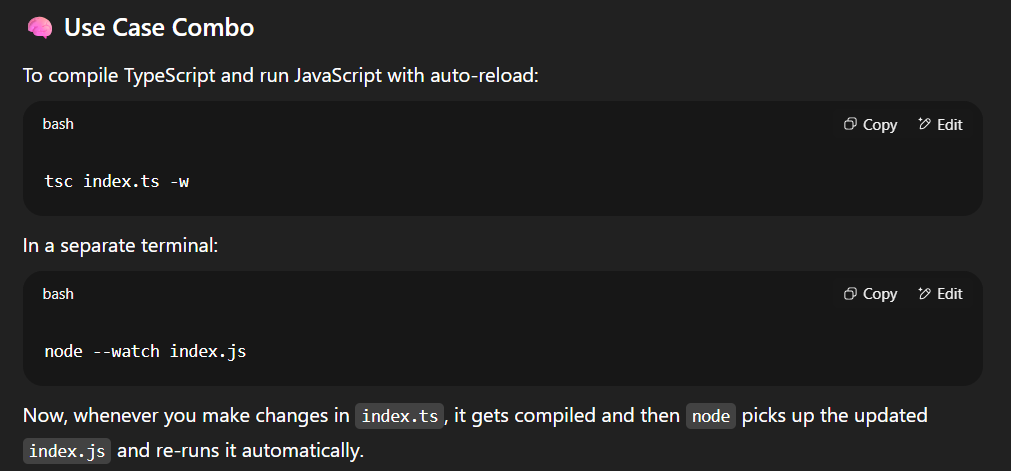
## Key Takeaway

> Instead of defining object shapes inline (`let user: {name: string}`), create a reusable `type` alias (`type User = {name: string}`). This makes your code cleaner and easier to maintain. Use `?` for optional properties.

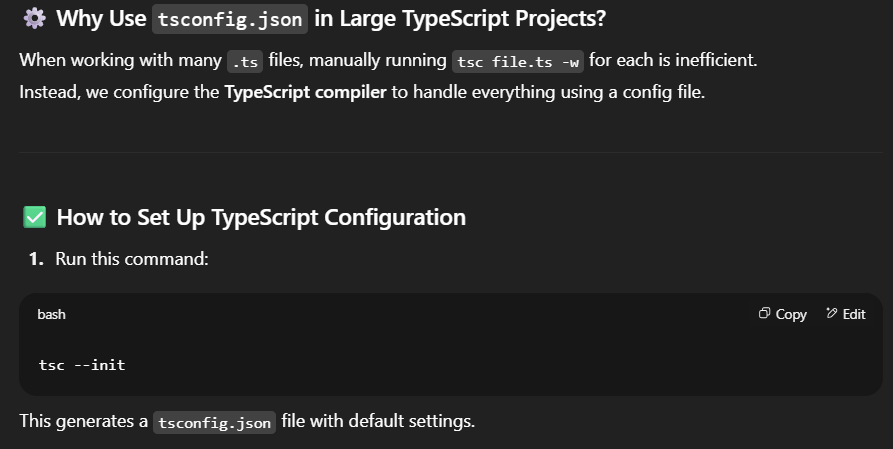
**Watch mode:**



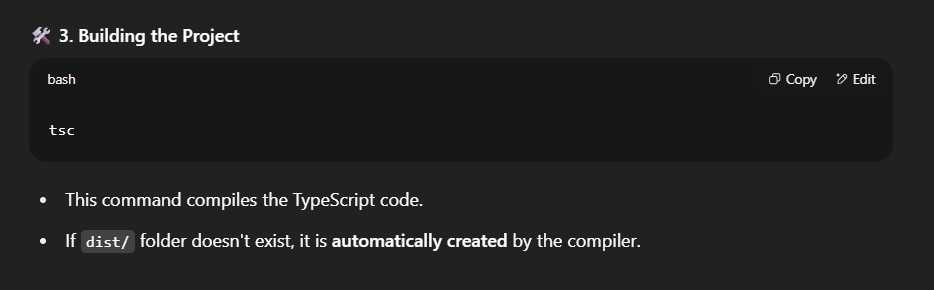
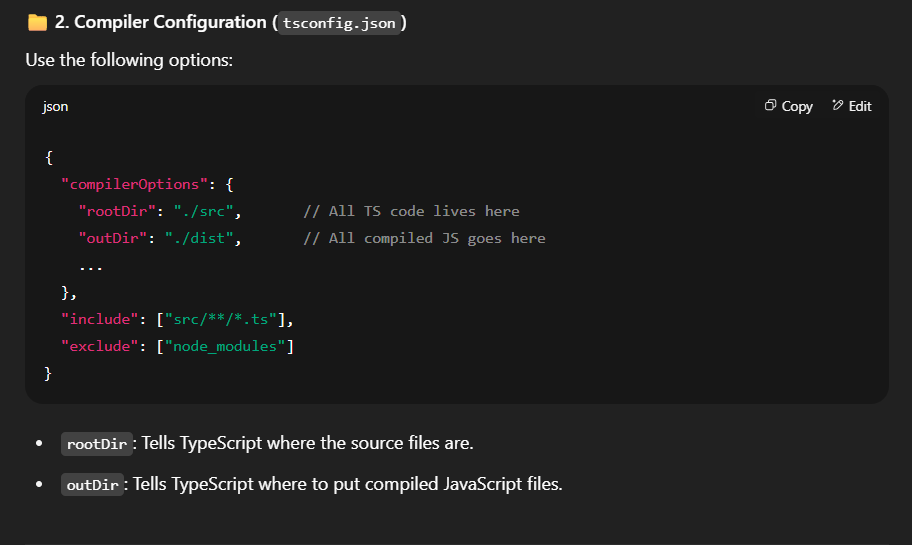
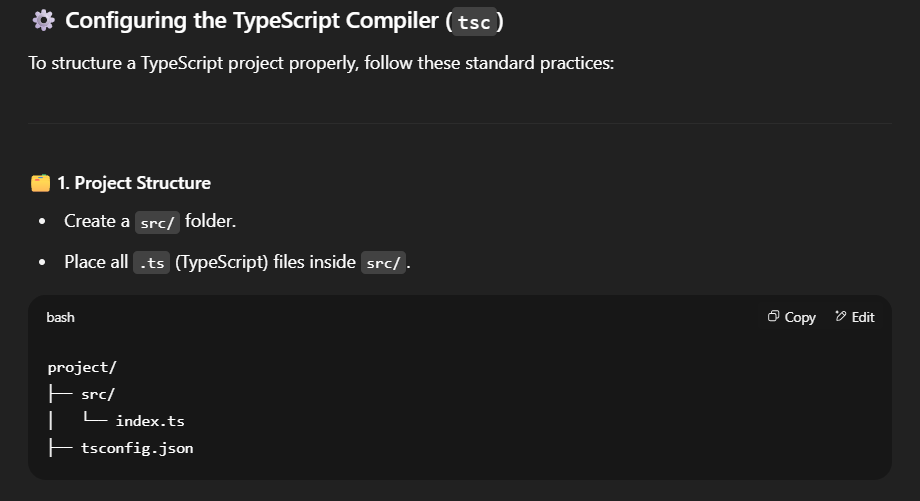


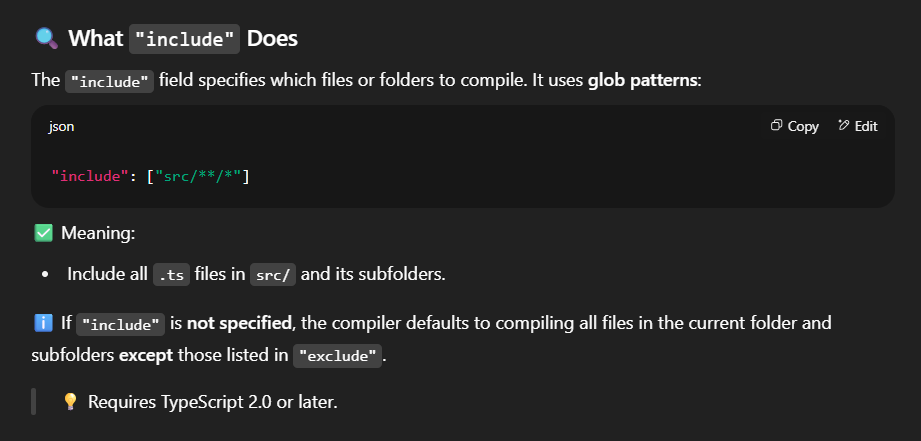


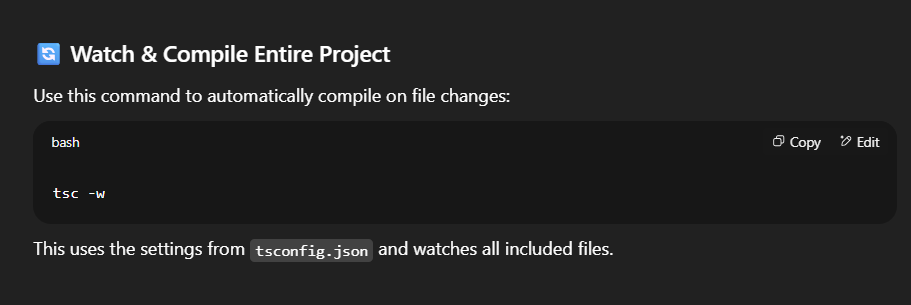
**TS compiler**

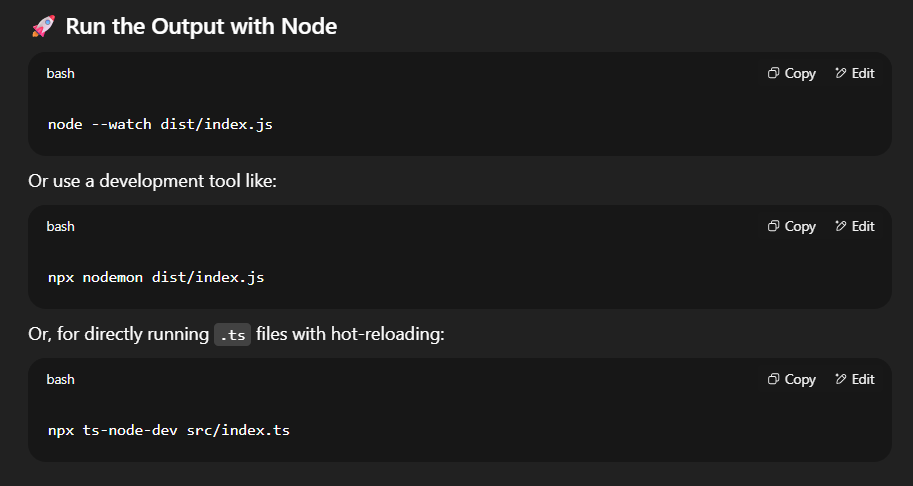


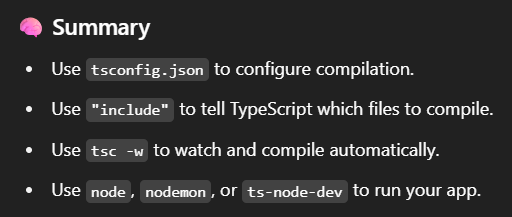




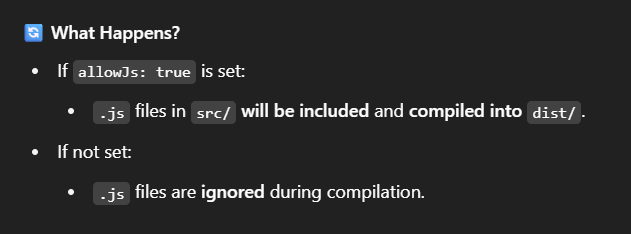
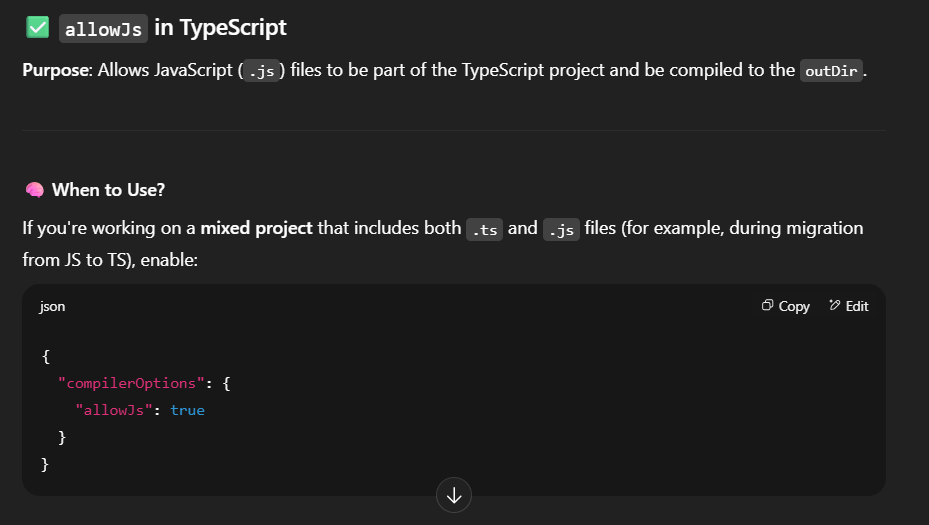




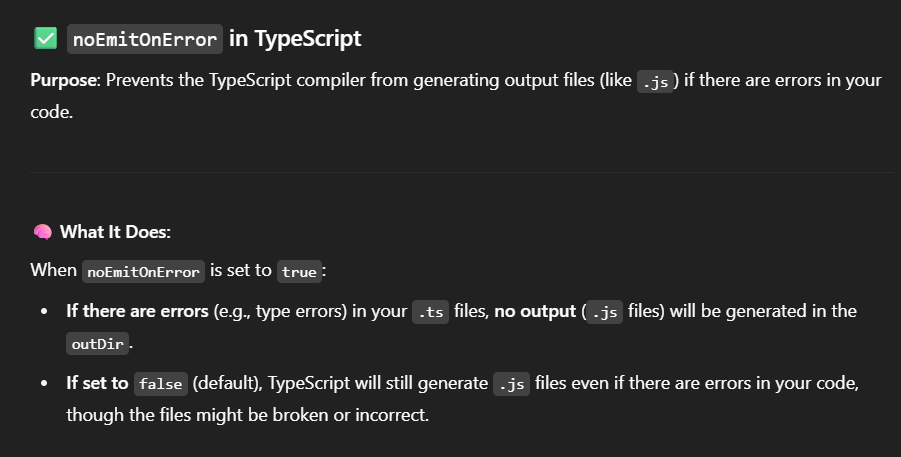


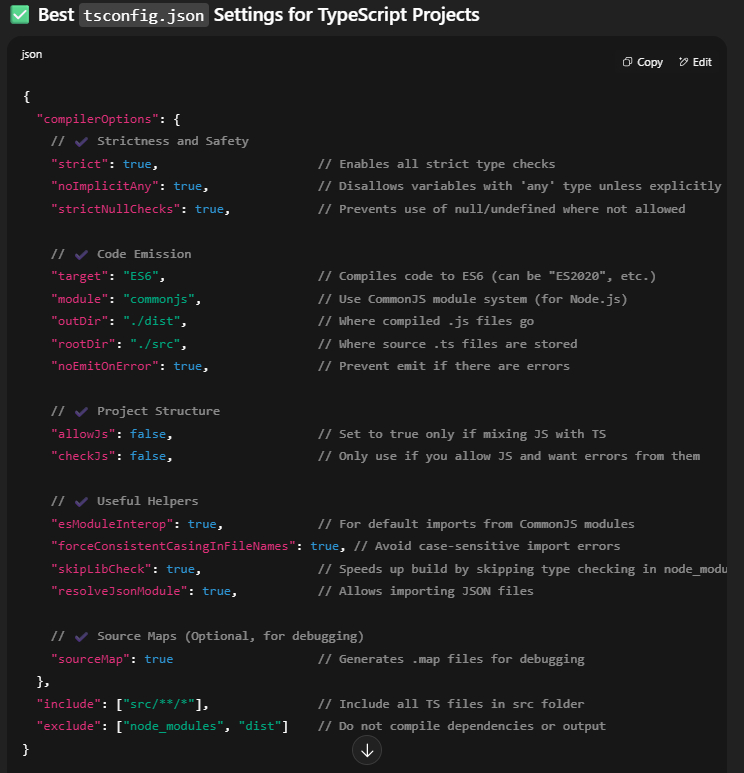


**AllowJS:**

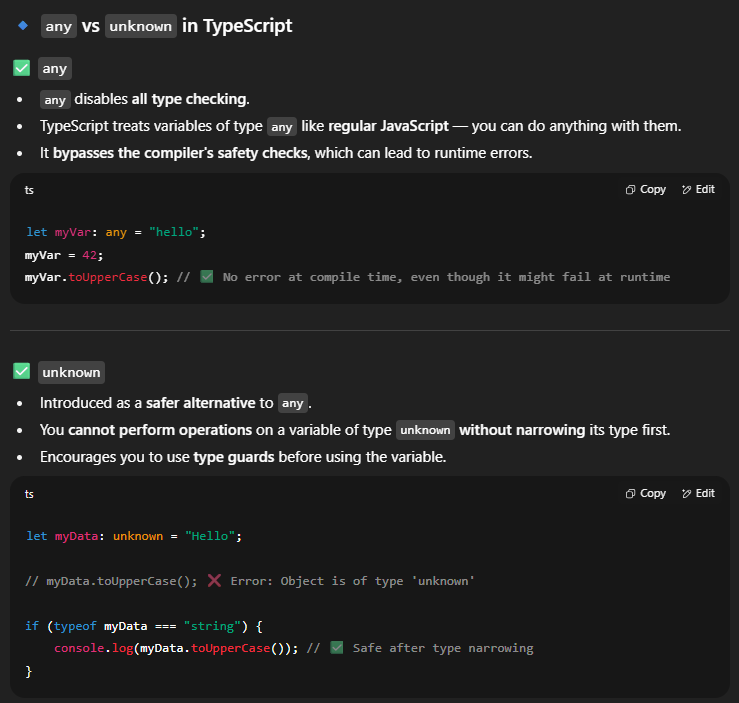


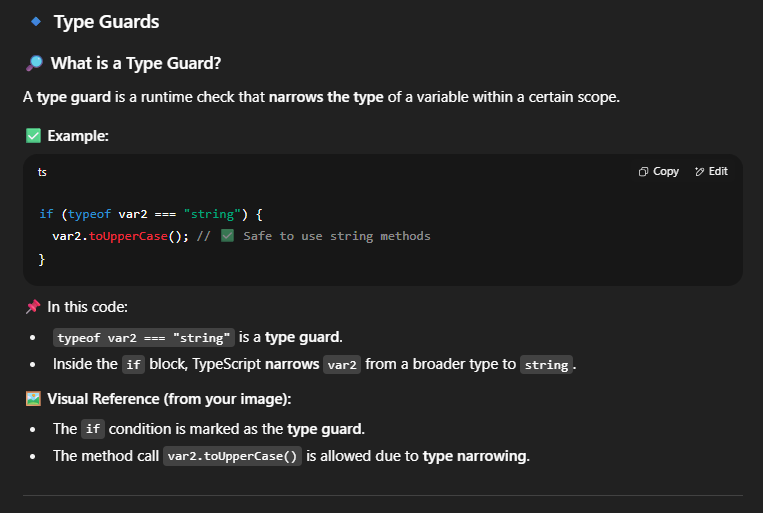
**noEmitOnError**

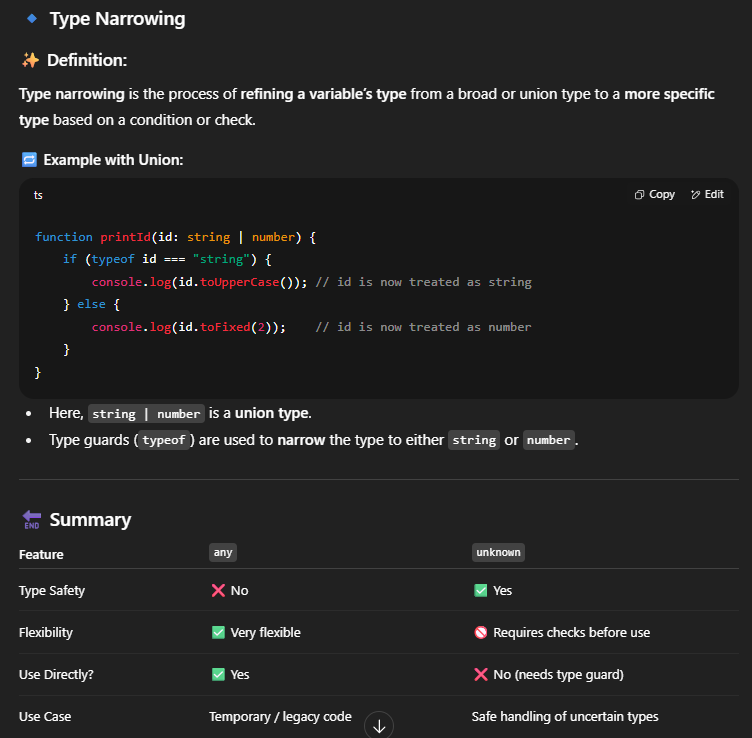




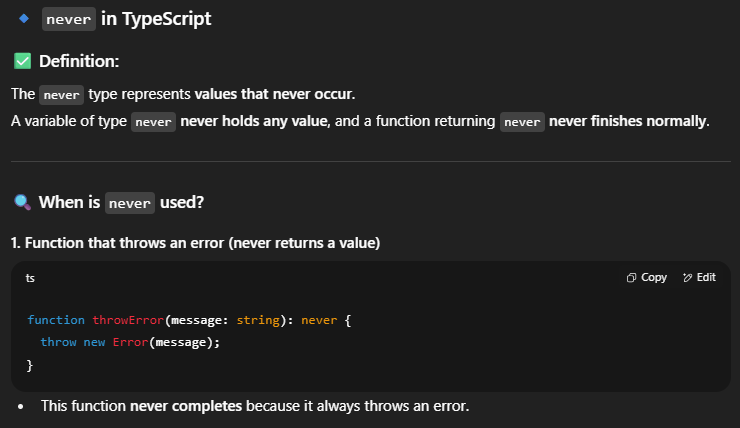
**Any vs Unkown**

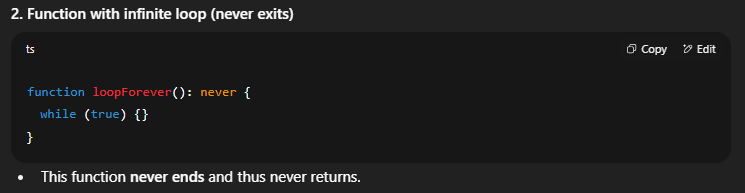


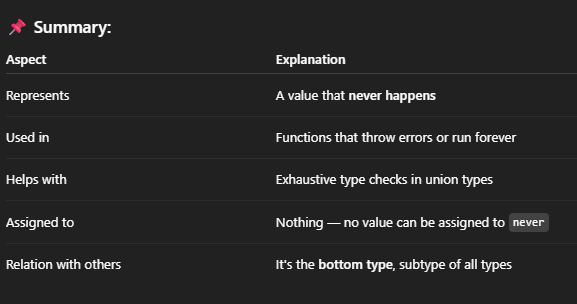




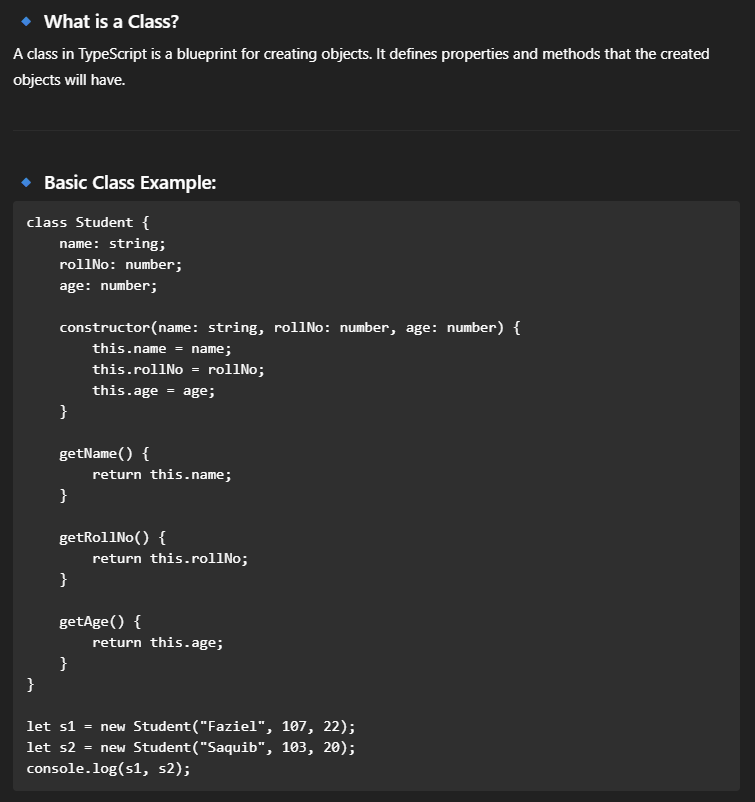
**Never**

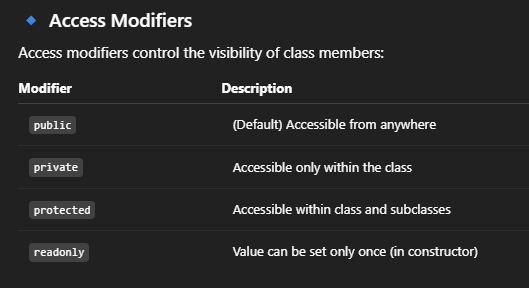
****

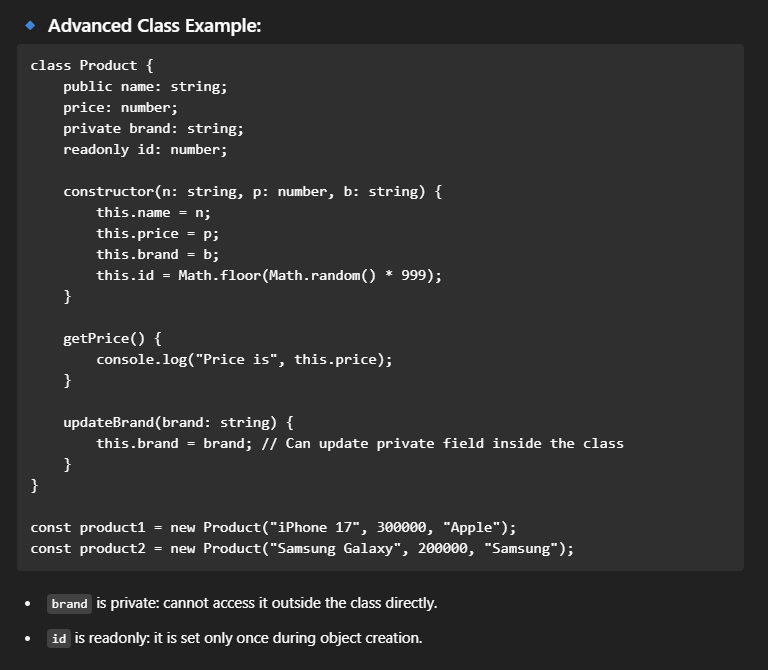
****

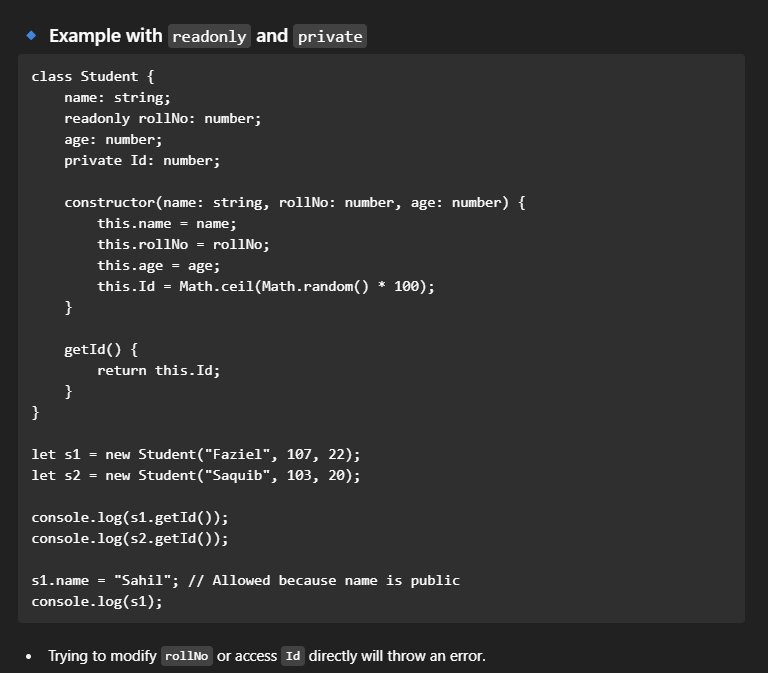
****

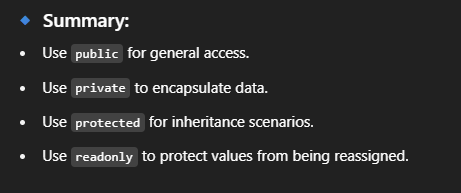
**Class in TS**

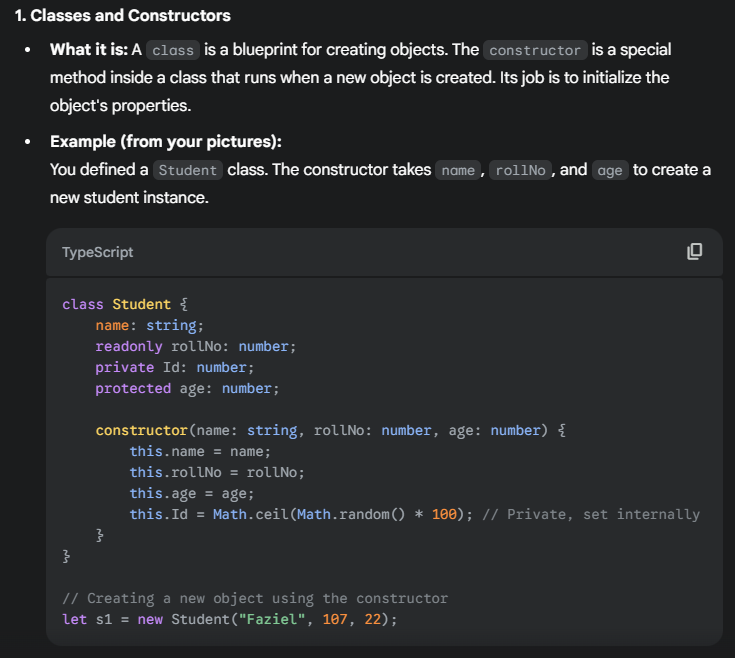
****

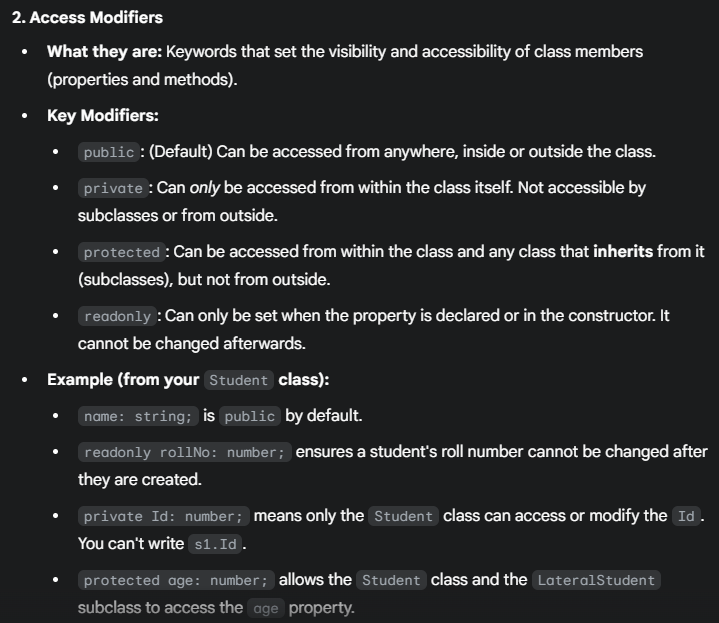
****

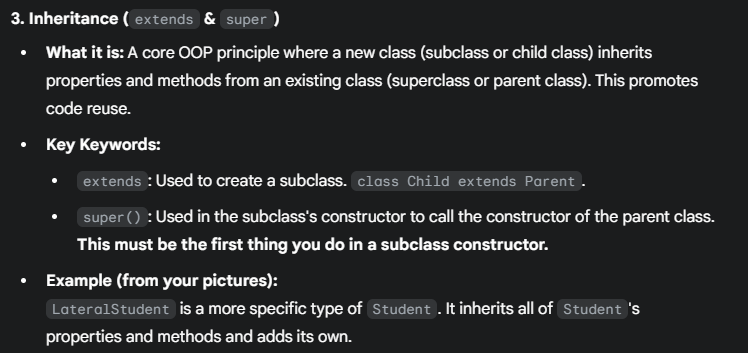
****

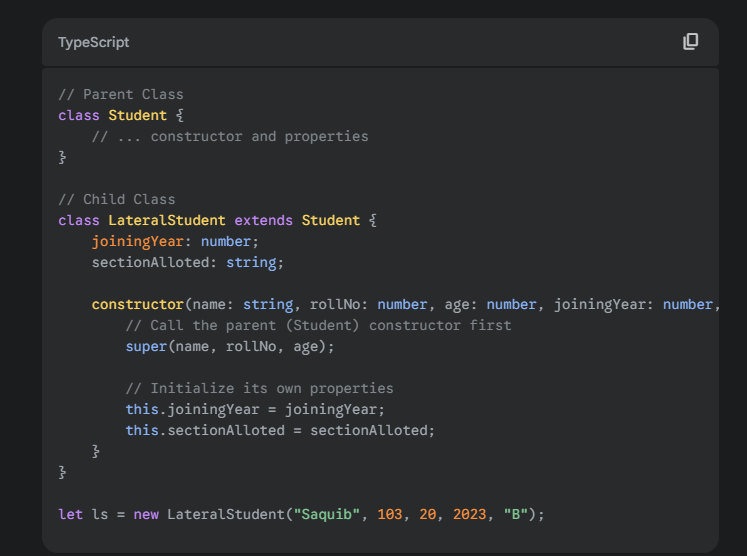
****

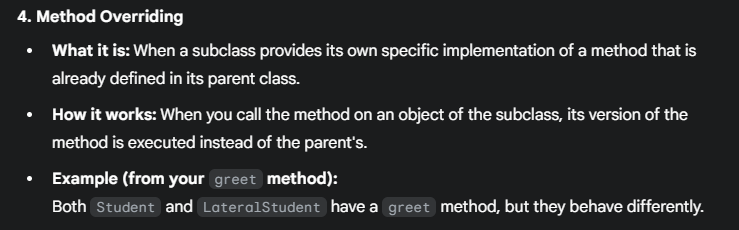
****

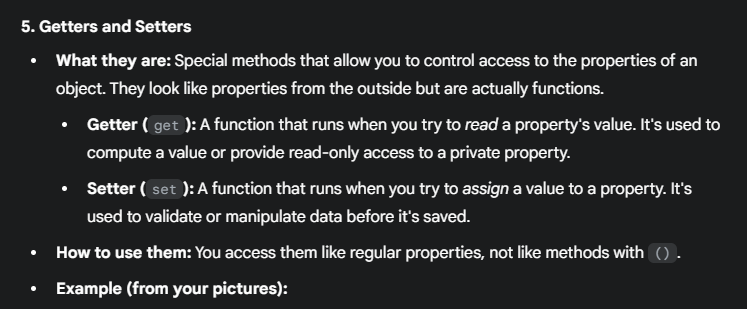
****

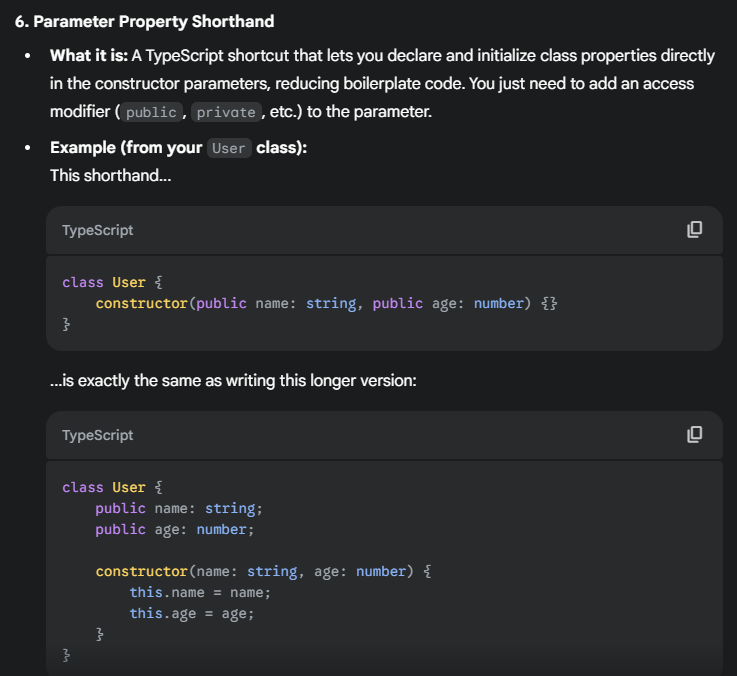
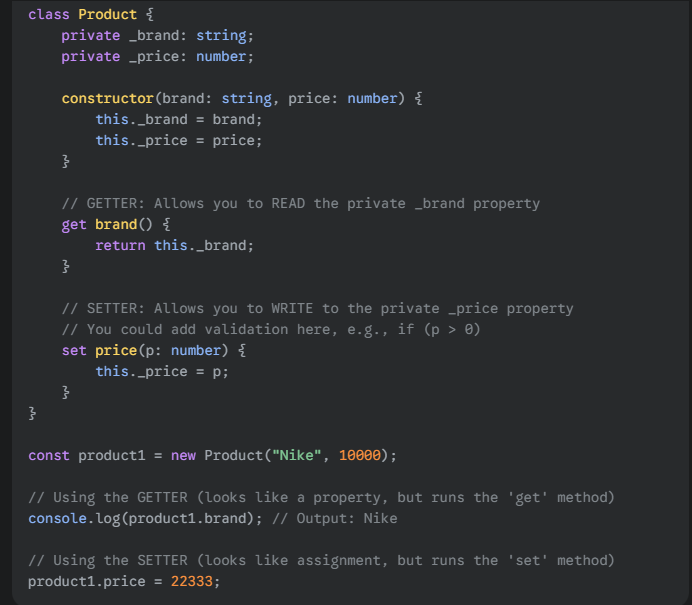
****

****

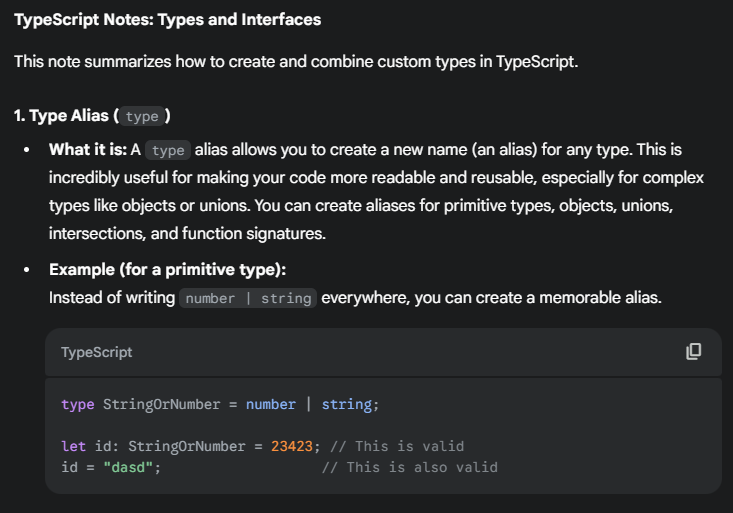
****

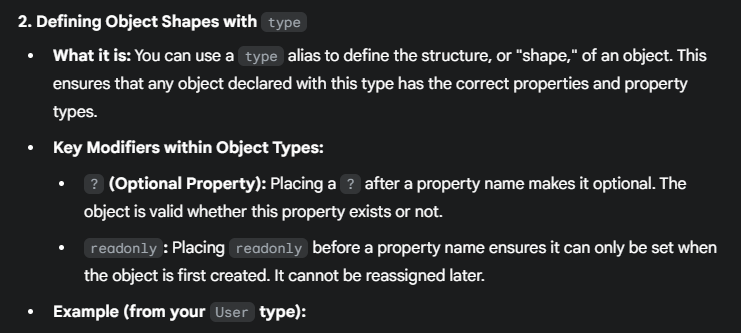
****

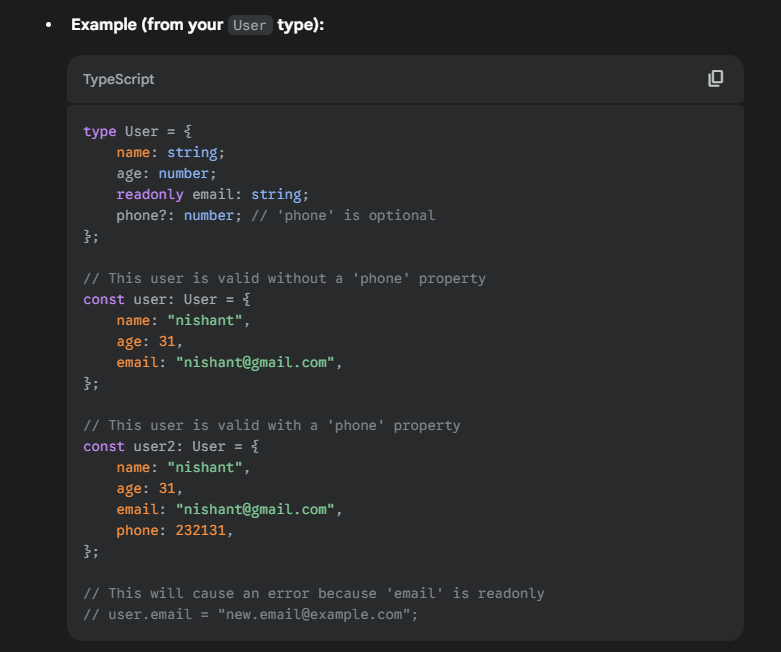
****

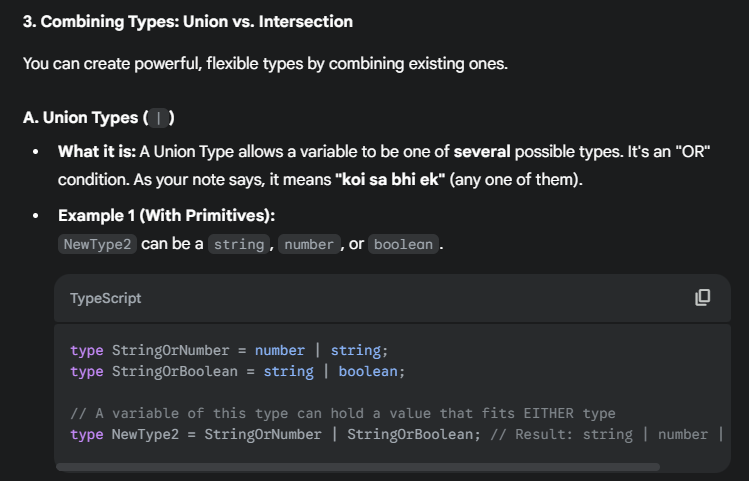
****

**Type Alias**

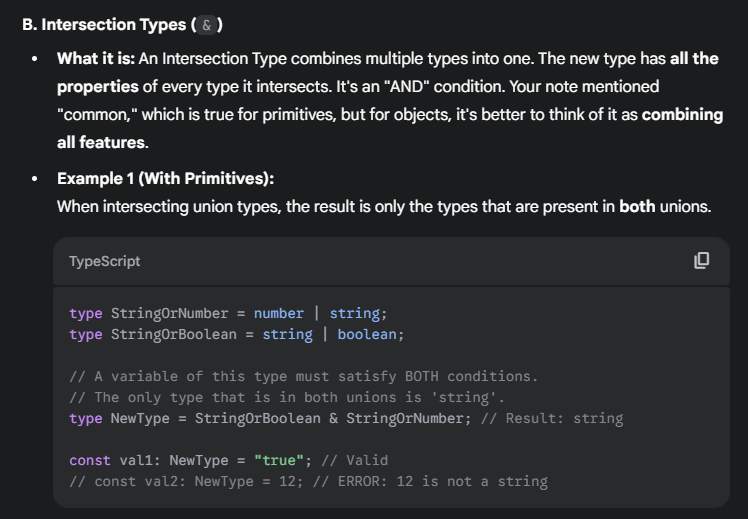
****

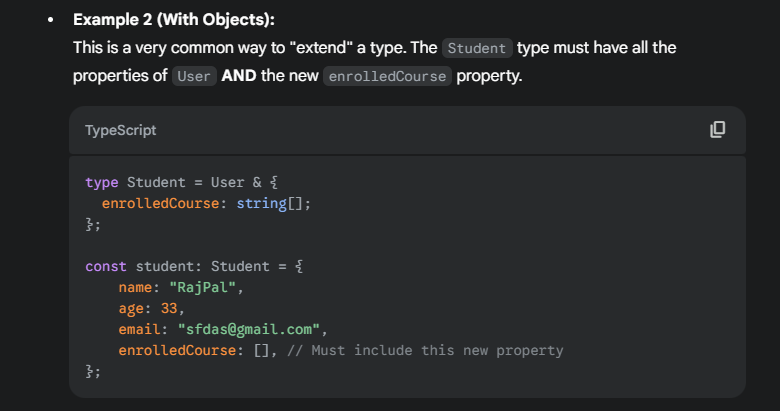
****

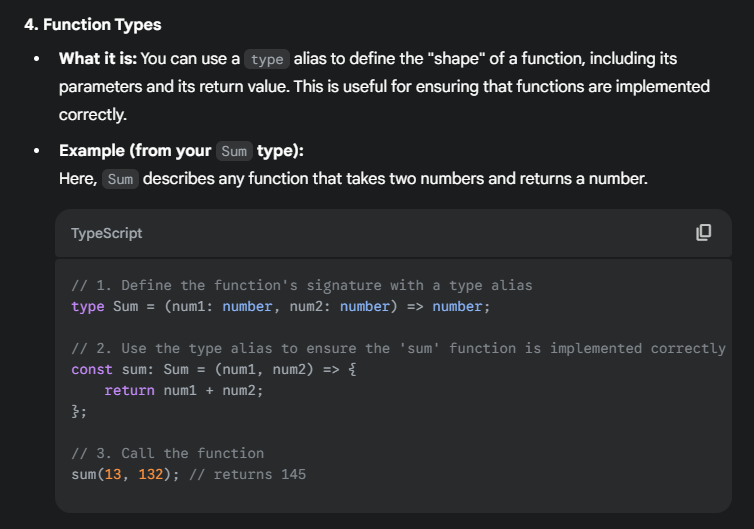
****

****

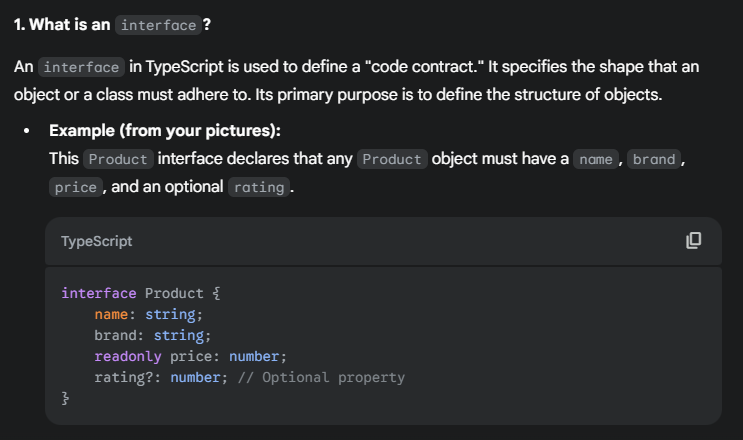
****

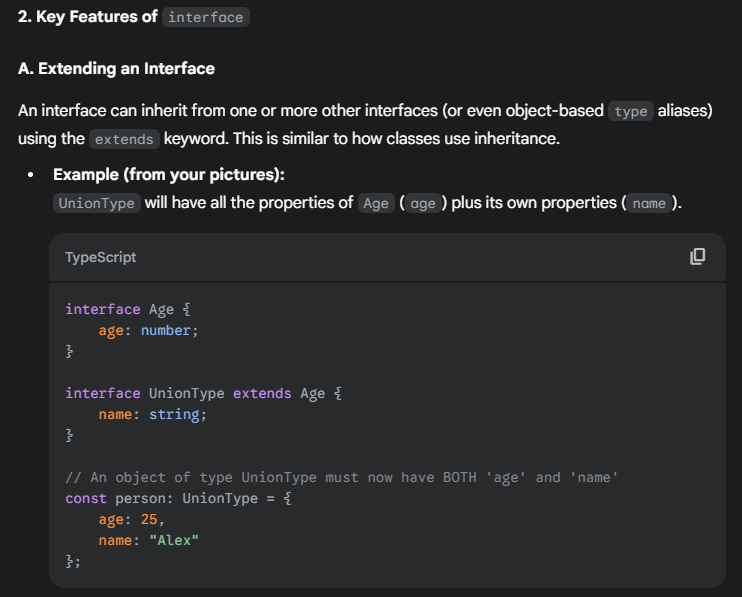
****

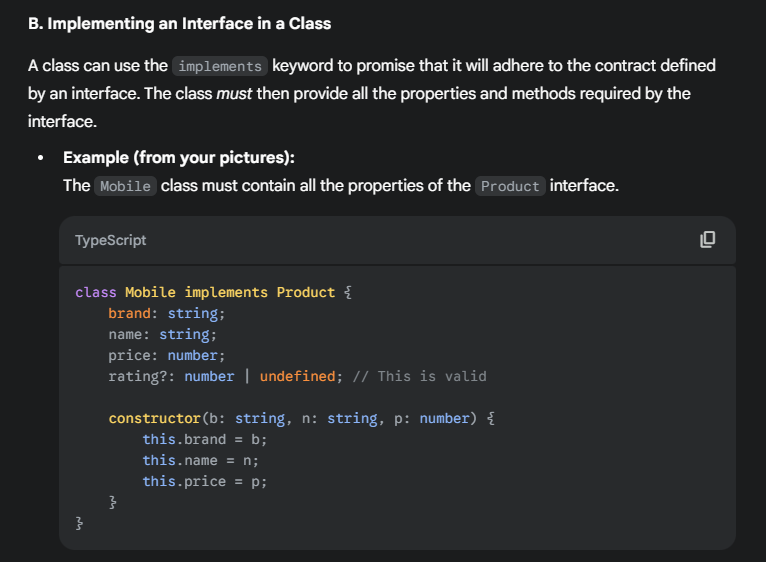
****

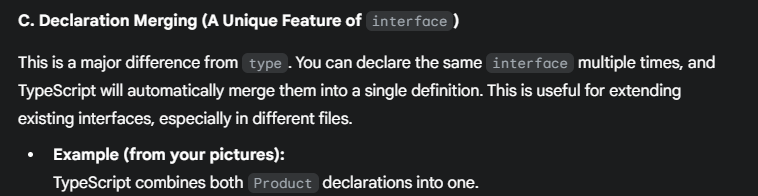
****

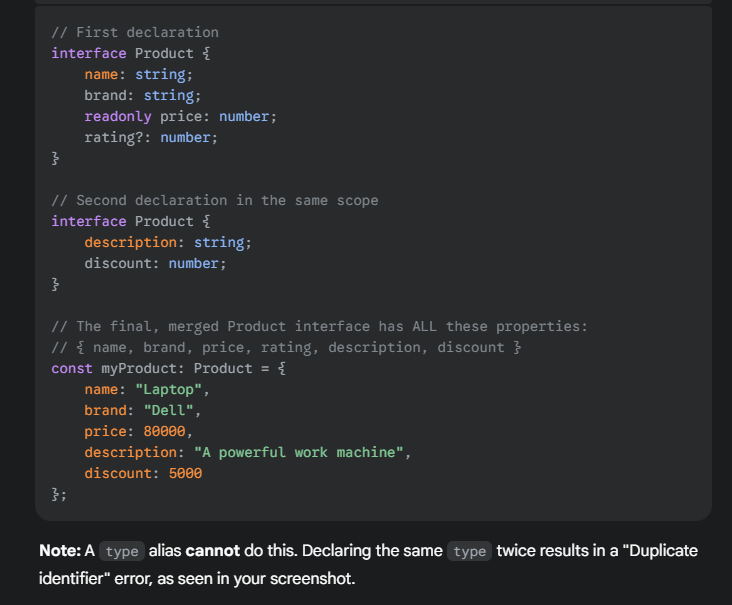
**Interface vs Type**

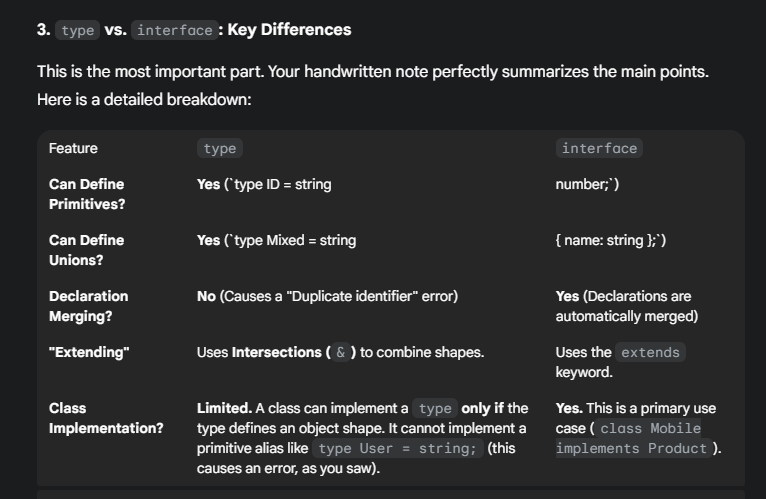
****

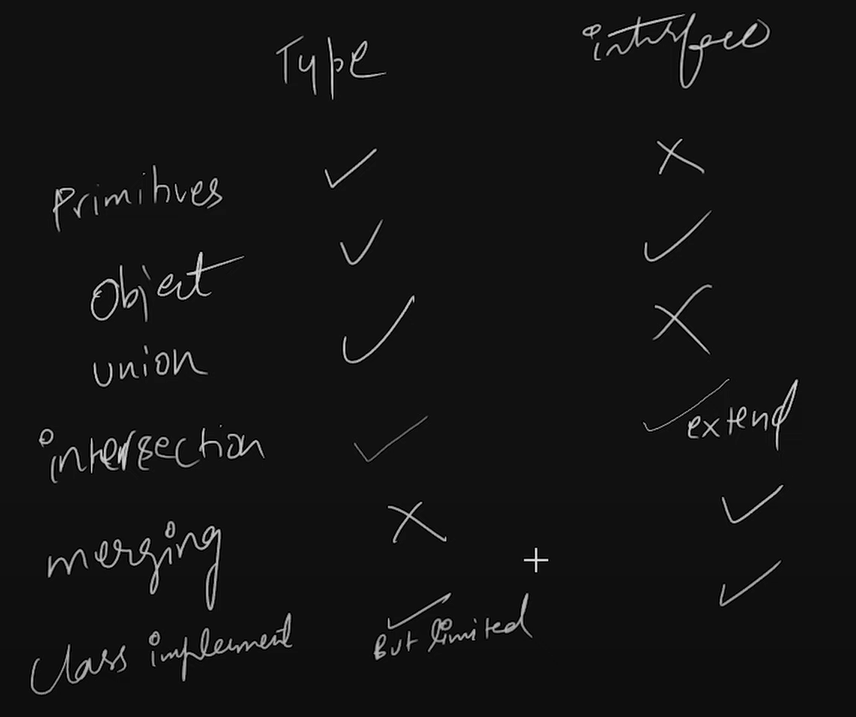
****

****

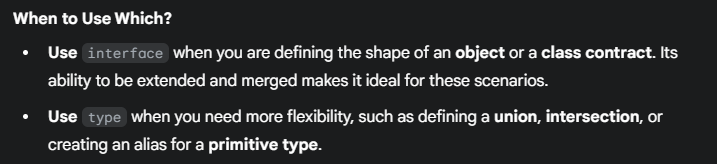
****

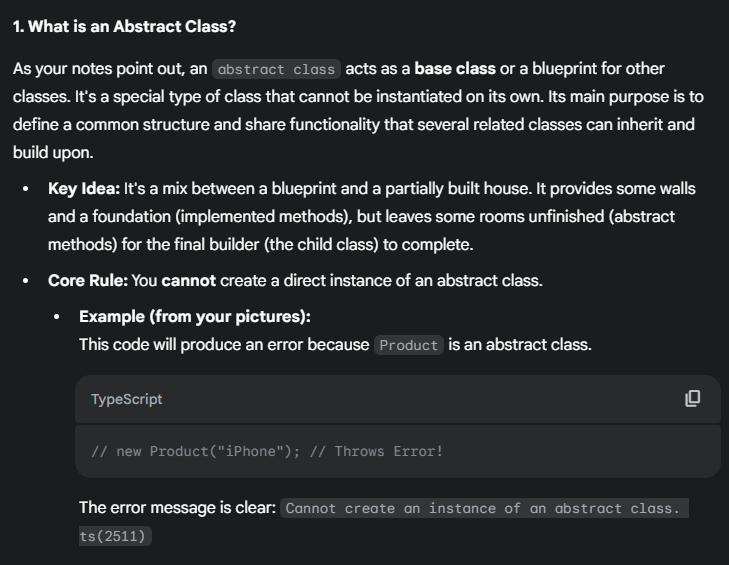
****

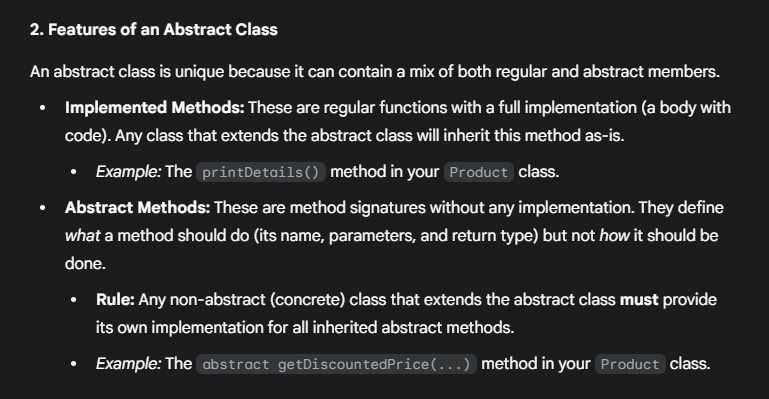
****

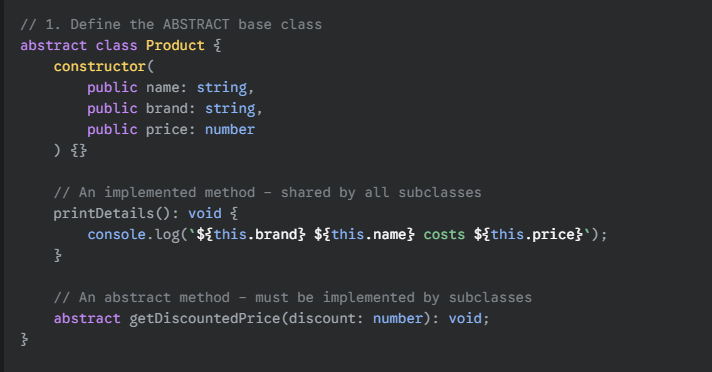


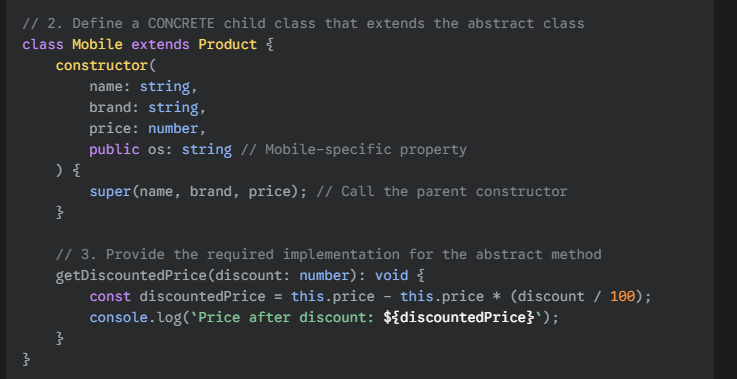
**Interface And Abstract Class**

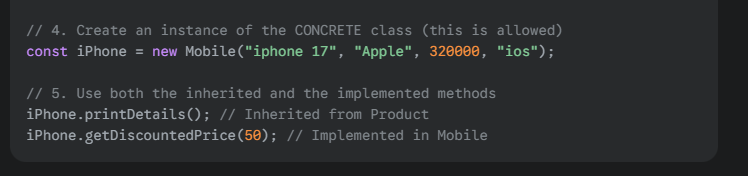
****

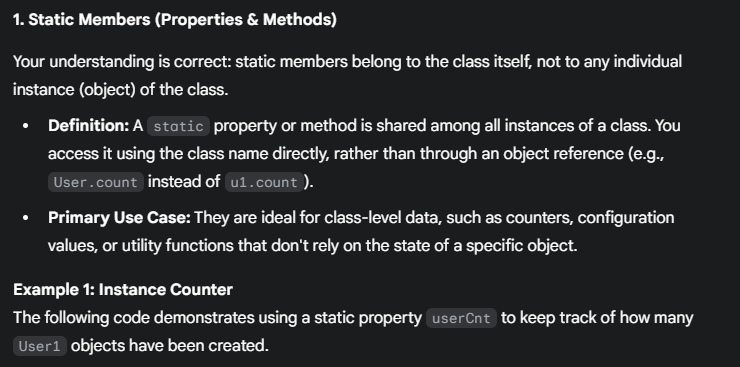
****

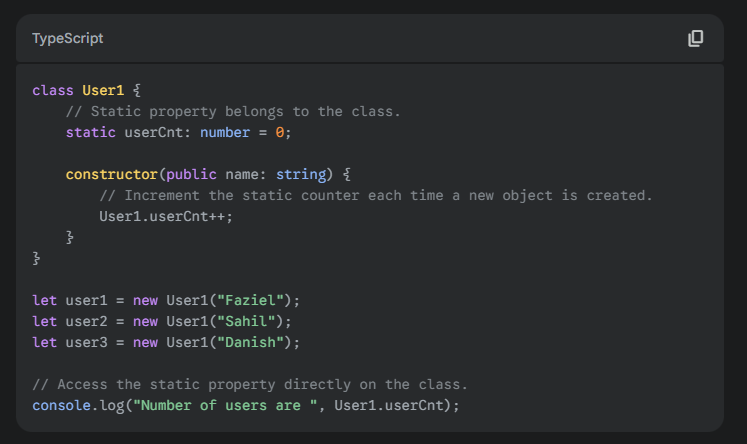
****

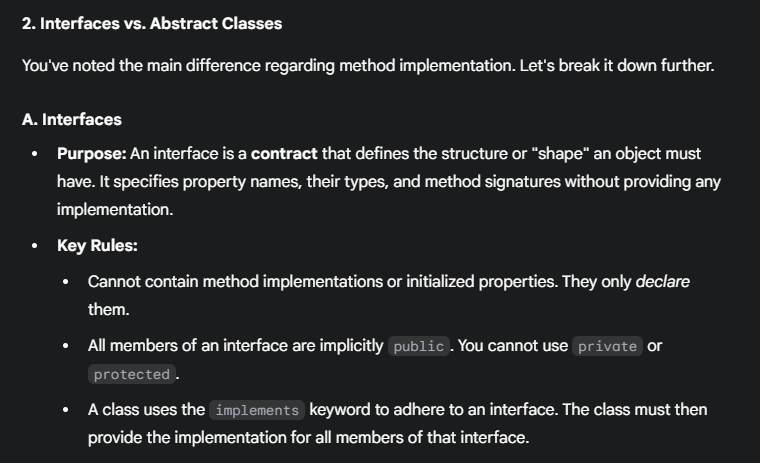
****

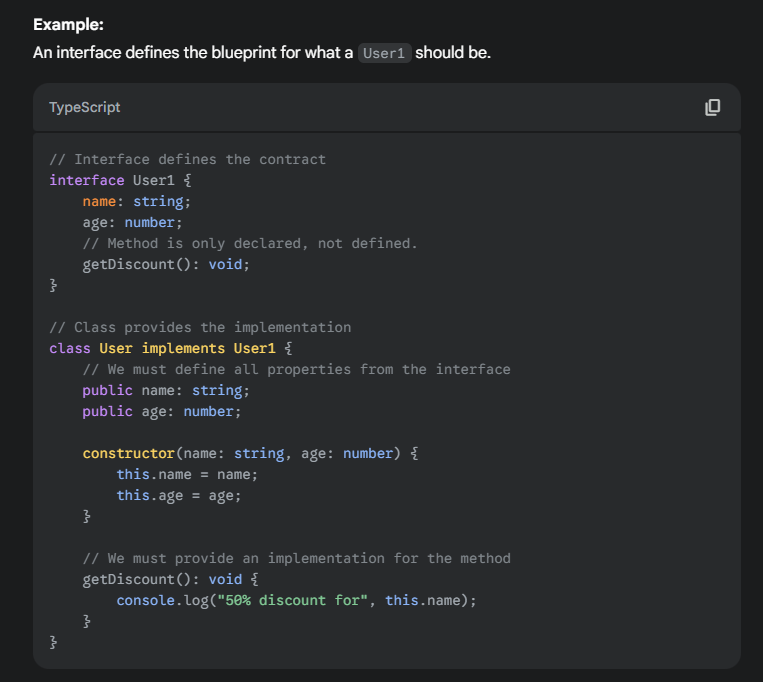
****

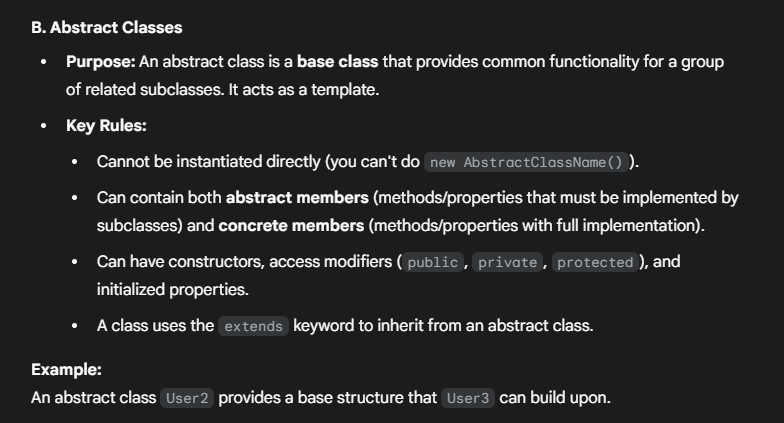
****

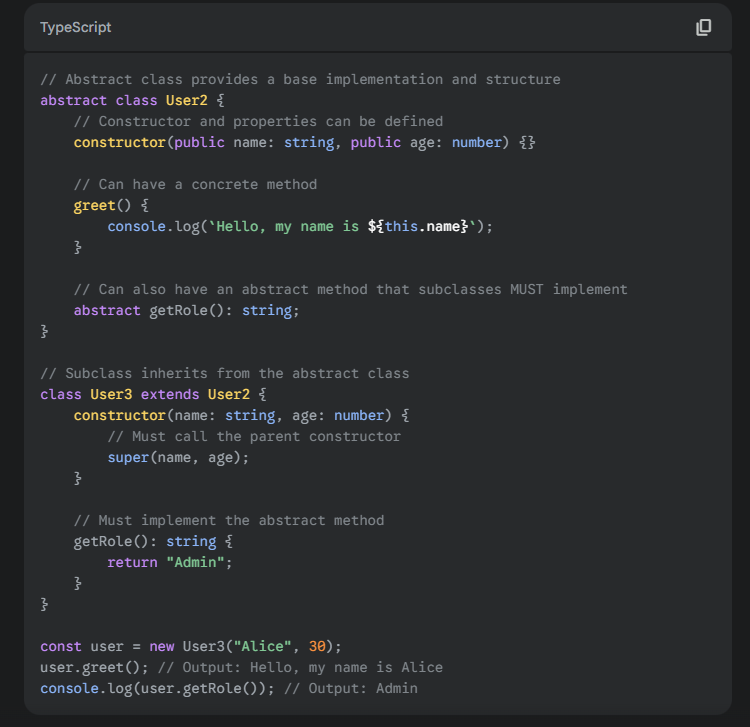
****

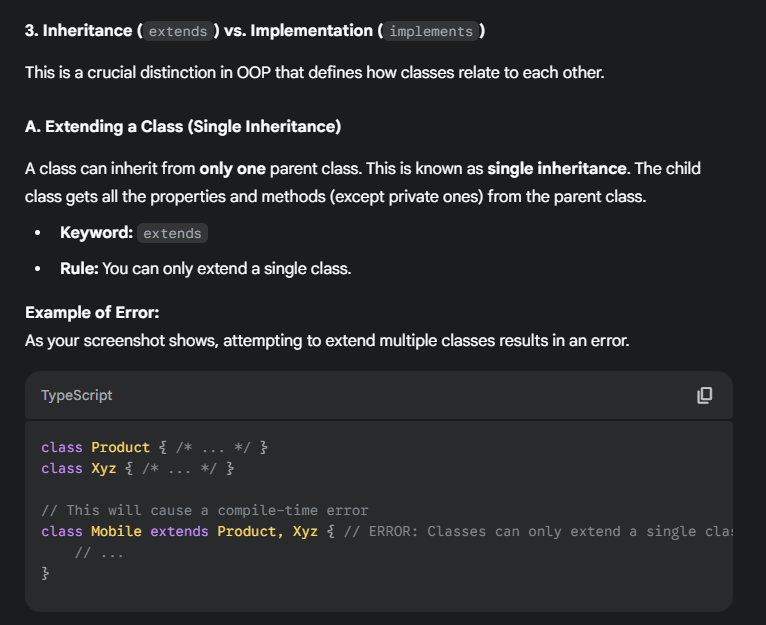
****

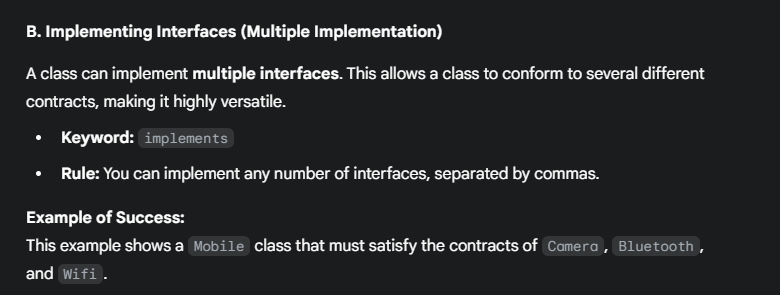
****

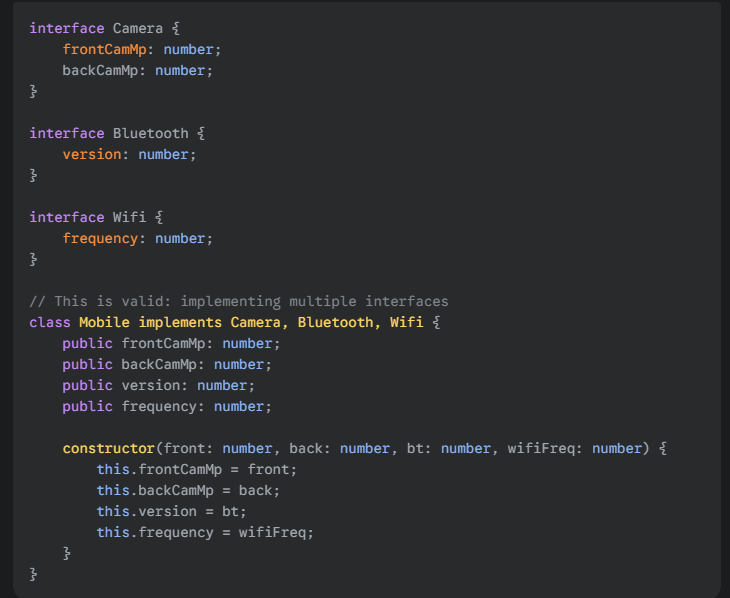
****

****

****

****

****

****