& Modules in TypeScript

What is an Interface?

An **interface** in TypeScript is a way to define the **structure of an object**. It tells the compiler what **properties and types** an object should have. It's like a **blueprint** for objects.

Interfaces help in:

- Ensuring **type safety**
- Making code more **readable and maintainable**
- Supporting **object-oriented programming**

Syntax

```
interface InterfaceName {
  property1: type;
  property2: type;
}
```

Example 1: Basic Interface

```
interface Person {
  name: string;
  age: number;
}

const student: Person = {
  name: "John",
  age: 22,
};
```

➡️ Here, the student object must match the structure defined in the Person interface.

Example 2: Optional Properties

Use ? to make a property optional.

```
interface Employee {
  id: number;
  name: string;
  department?: string; // optional
}

const emp: Employee = {
  id: 101,
  name: "Alice"
};
```

Example 3: Readonly Properties

Use readonly to prevent modification.

```
interface Book {
  readonly isbn: string;
  title: string;
}

const b1: Book = {
  isbn: "123-abc",
  title: "Learn TypeScript"
};

// b1.isbn = "456-def"; ❌ Error: Cannot assign to 'isbn' because it is a read-only property.
```

Example 4: Interface for Function Types

You can describe functions too!

```
interface Greet {
  (name: string): string;
}

const sayHello: Greet = (name) => {
  return `Hello, ${name}`;
};
```

Example 5: Extending Interfaces

Interfaces can be extended (like inheritance).

```
interface Animal {
  name: string;
}
interface Dog extends Animal {
  breed: string;
}
const myDog: Dog = {
  name: "Buddy",
  breed: "Labrador"
};
```

Summary

- Interfaces define the **structure of objects**.
- They support **optional**, **readonly**, and **function properties**.
- Interfaces can be **extended** for reusability.
- They improve **code clarity and maintainability**.

Differences Between Class and Interface in TypeScript

Feature	Class	Interface
Definition	Blueprint to create objects with **implementation**	Defines **structure/type** of an object with **no implementation**
Contains	Properties, constructors, methods (with implementation)	Only property and method **signatures** (no implementation)
Instantiation	Can be **instantiated** using new keyword	Cannot be instantiated directly
Inheritance	Supports **single inheritance** using extends	Supports **multiple inheritance** using extends
Implements	A class can **implement** one or more interfaces	Interface cannot implement a class
Modifiers	Supports access modifiers (public, private, protected)	Does **not** support access modifiers

Example for Clarification

Interface:

```
interface Animal {
  name: string;
  sound(): void;
}
```

Class:

```
class Dog implements Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  sound() {
    console.log("Woof!");
```

```
 }
}
const pet = new Dog("Tommy");
pet.sound(); // Woof!
```

Summary

- Use **interface** when you want to define the **structure or contract**.
- Use **class** when you want to define **implementation and create objects**.

Here's a beginner-friendly note on **TypeScript Modules** along with a simple example to demonstrate how they work.

TypeScript Modules

What are Modules?

Modules in TypeScript are a way to **organize and encapsulate code**. They help in dividing large programs into smaller, manageable files and allow reusability by exporting and importing functionalities (like variables, functions, classes, etc.).

A module in TypeScript is just a file containing **export** and/or **import** statements.

Why use Modules?

- Code reusability
- Better maintainability
- Avoid name conflicts using scoped declarations

How to Create and Use Modules?

- Use the export keyword to expose code from a file (module).
- Use the import keyword to bring in code from another file.

Example:

File: mathUtils.ts

```
// Exporting functions from this module
export function add(a: number, b: number): number {
```

```
    return a + b;
}
export function subtract(a: number, b: number): number {
    return a - b;
}
```

File: main.ts

```
// Importing functions from mathUtils module
import { add, subtract } from "./mathUtils";

console.log("Addition:", add(10, 5));      // Output: 15
console.log("Subtraction:", subtract(10, 5)); // Output: 5
```

Multiple ways to import modules:

1. Named Imports (Most common)

Used when the module exports multiple things using export.

Module.ts

```
export const AppName = "MyApp";
export function add(a: number, b: number): number {
   return a + b;
}
```

Usage:

```
import { AppName, add } from "./Module";
```

2. Aliased Imports

You can rename imports for clarity or conflict avoidance.

```
import { add as addNumbers, AppName as NameOfApp } from "./Module";
```

3. Import Everything (Namespace Import)

If you want to import the whole module under a single object.

```
import * as Utils from "./Module";


console.log(Utils.add(2, 3));
console.log(Utils.AppName);
```

This is useful when you want to group everything from a module or avoid listing individual exports.