Typescript

Basics Typescript

📌 What is Typescript?

=> TypeScript is a superset of JavaScript that adds static typing and modern features to make code more robust and maintainable. It compiles to plain JavaScript, ensuring compatibility with all browsers and environments.

```
👉 Simply put: TypeScript = JavaScript + Type Safety + Better Tooling.

👉Typescript is a development tool.The main project still run on js.
```

📌 JS vs TS

Feature	JavaScript (JS)	TypeScript (TS)
Type System	Dynamically typed (no static type checking)	Statically typed (catches errors at compile time)
Compilation	Runs directly in browsers	Needs to be compiled into JS using `tsc`
Error Detection	Errors appear at runtime	Errors detected during development (before execution)
Object-Oriented Features	Supports classes (ES6), but lacks interfaces	Supports interfaces, generics, and access modifiers
Code Maintainability	Can get messy in large projects	Better for large-scale applications due to type safety
Tooling Support	Basic code completion	Advanced autocomplete, IntelliSense, and better refactoring tools
Community Support	Widely used, native to web	Growing rapidly, used in enterprise applications

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

TypeScript Installation Guide

📌 Prerequisites

Before installing TypeScript, ensure you have **Node.js** installed on your system.

☑ Check if Node.js is Installed

Run the following command in the terminal:

```
node -v
```

If Node.js is not installed, download and install it from Node.js Official Website.

📌 Installing TypeScript

1️⃣ Global Installation (Recommended)

This will install TypeScript globally so you can use the `tsc` (TypeScript Compiler) command anywhere.

```
npm install -g typescript
```

2️⃣ Verify Installation

Check if TypeScript is installed correctly:

```
tsc --version
```

If installed, it will display the TypeScript version, e.g., `Version 5.2.3`

3️⃣ Installing TypeScript Locally in a Project

For project-specific installation, use:

```
npm install --save-dev typescript
```

This will add TypeScript as a `devDependency` in `package.json`.

📌 Setting Up TypeScript in a Project

1️⃣ Initialize a TypeScript Project

Run the following command to generate a `tsconfig.json` file:

```
tsc --init
```

This file allows you to configure TypeScript settings.

2️⃣ Compile a TypeScript File

Create a sample TypeScript file `index.ts`:

```
const greet = (name: string): string => {
    return `Hello, ${name}!`;
};
console.log(greet("Anik"));
```

Compile it using:

```
tsc index.ts
```

This will generate a JavaScript file `index.js`, which can be run using:

```
node index.js
```

📌 TypeScript with VS Code

- Install the TypeScript extension in VS Code.
- Enable Auto Compilation by running:

```
tsc --watch
```

This will automatically compile TypeScript files when changes are made.

📌 TypeScript with Popular Frameworks

- **React:**

```
npx create-react-app my-app --template typescript
```

- **Node.js:**

```
npm install -g ts-node
```

🔥 Types of TypeScript

TypeScript provides a strong type system that enhances JavaScript by introducing various types.

Syntax:

```
let variableName : type = value
```

Below are the main types in TypeScript:

📌 1. Primitive Types

Primitive types are basic data types that represent simple values.

☑ String

```ts
let name: string = "Anik";
```

☑ Number

```ts
let age: number = 20;
```

☑ Boolean

```ts
let isDeveloper: boolean = true;
```

☑ Null & Undefined

```ts
let value: null = null;
let notAssigned: undefined = undefined;
```

📌 2. Object Types

Object types represent complex data structures.

☑ Object

```ts
let user: { name: string; age: number } = { name: "Anik", age: 20 };
```

☑ Array

```
let numbers: number[] = [1, 2, 3, 4, 5];
let names: Array<string> = ["Anik", "John", "Doe"];
```

✅ Tuple (Fixed-length array)

```
let person: [string, number] = ["Anik", 20];
```

✅ Enum

```
enum Role {
  Admin,
  User,
  Guest
}
let myRole: Role = Role.Admin;
```

📌 3. Special Types

Some special types help in flexibility and type safety.

✅ Any (Disables Type Checking)

```
let randomValue: any = "Hello";
randomValue = 42;
```

✅ Unknown (Safer Alternative to Any)

```
let unknownValue: unknown = "Hello";
if (typeof unknownValue === "string") {
  console.log(unknownValue.toUpperCase());
}
```

✅ Void (Used for Functions Without Return Value)

```
function logMessage(): void {
  console.log("Hello, TypeScript!");
}
```

✅ Never (Represents Values That Never Occur)

```
function throwError(message: string): never {
  throw new Error(message);
}
```

📌 4. Advanced Types

✅ Union (Multiple Possible Types)

```
let id: string | number;
id = "123";
id = 123;
```

✅ Intersection (Combining Types)

```
type Person = { name: string };
type Employee = { employeeId: number };
type Worker = Person & Employee;
let worker: Worker = { name: "Anik", employeeId: 101 };
```

✅ Type Aliases

```
type Point = { x: number; y: number };
let p: Point = { x: 10, y: 20 };
```

✅ Literal Types

```
let direction: "up" | "down";
direction = "up";
```

TypeScript Functions

TypeScript provides function types that enhance type safety and readability.

📌 Basic Function

```
function greet(name: string): string {
  return `Hello, ${name}!`;
}
console.log(greet("Anik"));
```

📌 Function with Optional Parameters

```
function greet(name: string, age?: number): string {
  return age ? `Hello, ${name}, you are ${age} years old!` : `Hello, ${name}!`;
}
console.log(greet("Anik"));
console.log(greet("Anik", 20));
```

📌 Function with Default Parameters

```
function greet(name: string = "Guest"): string {
  return `Hello, ${name}!`;
}
console.log(greet());
console.log(greet("Anik"));
```

📌 Function with Rest Parameters

```
function sum(...numbers: number[]): number {
  return numbers.reduce((acc, num) => acc + num, 0);
}
console.log(sum(1, 2, 3, 4));
```

📌 Arrow Function

```
const add = (a: number, b: number): number => a + b;
console.log(add(10, 20));
```

📌 Function Overloading

```
function display(value: string): void;
function display(value: number): void;
function display(value: any): void {
  console.log(value);
}
```

```
display("Hello");
display(123);
```

Bad Behavior of Objects in TypeScript

Objects in TypeScript can sometimes exhibit unexpected behaviors due to JavaScript's underlying nature. Here are some common pitfalls and how to handle them.

📌 1. Object Mutability

By default, objects in TypeScript (and JavaScript) are mutable, which can lead to unintended modifications.

❌ Bad Practice

```
const user = { name: "Anik", age: 20 };
user.age = 25; // Mutates the original object
console.log(user); // { name: "Anik", age: 25 }
```

✅ Best Practice: Use Readonly

```
type User = Readonly<{ name: string; age: number }>;
const user: User = { name: "Anik", age: 20 };
// user.age = 25; // Error: Cannot assign to 'age' because it is a read-only property.
```

📌 2. Object Reference Issues

Objects are reference types, meaning modifying a copy also affects the original.

❌ Bad Practice

```
const person1 = { name: "Anik" };
const person2 = person1;
person2.name = "John";
console.log(person1.name); // "John" (Unexpected change)
```

✅ Best Practice: Use Object Spread or `Object.assign()`

```
const person1 = { name: "Anik" };
const person2 = { ...person1 };
```

```
person2.name = "John";
console.log(person1.name); // "Anik" (No unintended modification)
```

📌 3. Comparing Objects

Objects are compared by reference, not by value.

❌ Bad Practice

```
const obj1 = { id: 1 };
const obj2 = { id: 1 };
console.log(obj1 === obj2); // false (Even though values are the same)
```

☑️ Best Practice: Compare Properties

```
const obj1 = { id: 1 };
const obj2 = { id: 1 };
console.log(JSON.stringify(obj1) === JSON.stringify(obj2)); // true
```

📌 4. Accidental Undefined Properties

Accessing non-existent properties leads to `undefined`.

❌ Bad Practice

```
const user = { name: "Anik" };
console.log(user.age); // undefined
```

☑️ Best Practice: Use Optional Chaining or Type Safety

```
type User = { name: string; age?: number };
const user: User = { name: "Anik" };
console.log(user.age ?? "Age not provided"); // "Age not provided"
```

📌 5. Modifying Function Parameters (Object Mutation)

Passing objects to functions without precautions can cause unexpected changes.

❌ Bad Practice

```
function updateAge(user: { name: string; age: number }) {
  user.age = 30;
}
const person = { name: "Anik", age: 20 };
updateAge(person);
console.log(person.age); // 30 (Unexpected modification)
```

☑ Best Practice: Pass a Copy

```
function updateAge(user: { name: string; age: number }) {
  return { ...user, age: 30 };
}
const person = { name: "Anik", age: 20 };
const updatedPerson = updateAge(person);
console.log(person.age); // 20 (Original remains unchanged)
console.log(updatedPerson.age); // 30
```

🚀 Conclusion

Understanding these bad behaviors and best practices ensures that objects in TypeScript behave predictably, reducing bugs and improving maintainability.

TypeScript Type Aliases

Type aliases in TypeScript allow you to create custom names for types, making your code more readable and reusable.

📌 Defining Type Aliases

A type alias is created using the `type` keyword.

☑ Basic Type Alias

```
// Defining a type alias for a string
type Username = string;

let user: Username = "Anik";
```

☑ Object Type Alias

```typescript
type Person = {
  name: string;
  age: number;
};

let person: Person = { name: "Anik", age: 20 };
```

📌 Using Type Aliases with Union Types

Type aliases can be used with union types to specify multiple possible types.

```typescript
type ID = string | number;

let userId: ID;
userId = "123ABC";
userId = 456;
```

📌 Using Type Aliases with Intersection Types

Intersection types allow combining multiple types into one.

```typescript
type Employee = { id: number; department: string };
type User = { name: string; age: number };

type EmployeeDetails = Employee & User;

let employee: EmployeeDetails = {
  id: 101,
  department: "IT",
  name: "Anik",
  age: 20
};
```

📌 Type Alias with Function Signatures

You can define function signatures using type aliases.

```typescript
type GreetFunction = (name: string) => string;

const greet: GreetFunction = (name) => `Hello, ${name}!`;
console.log(greet("Anik"));
```

📌 Type Alias with Tuples

Type aliases can also define tuples for fixed-length arrays.

```
type Coordinates = [number, number];

let point: Coordinates = [10, 20];
```

TypeScript Arrays

TypeScript provides strong typing for arrays, allowing better control and safety when handling collections of data.

📌 Defining Arrays

☑ Basic Array Syntax

You can define an array using the following syntax:

```
let numbers: number[] = [1, 2, 3, 4, 5];
let names: string[] = ["Anik", "John", "Doe"];
```

☑ Using the `Array<T>` Generic Type

```
let ids: Array<number> = [101, 102, 103];
let fruits: Array<string> = ["Apple", "Banana", "Cherry"];
```

📌 Readonly Arrays

A `readonly` array ensures elements cannot be modified after initialization.

```
let readonlyNumbers: readonly number[] = [10, 20, 30];
// readonlyNumbers.push(40); // ❌ Error: Property 'push' does not exist on type
'readonly number[]'
```

📌 Array of Objects

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

let users: User[] = [
    { id: 1, name: "Anik" },
    { id: 2, name: "John" }
];
```

📌 Multi-Dimensional Arrays

You can define multi-dimensional arrays like this:

```
let matrix: number[][] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

📌 Tuple Arrays

A **tuple** is a fixed-length array with specific types for each position.

```
let userInfo: [string, number] = ["Anik", 25];
```

📌 Array Methods in TypeScript

```
let numbers: number[] = [1, 2, 3, 4, 5];
console.log(numbers.length); // Output: 5

numbers.push(6); // Adds 6 to the end
numbers.pop(); // Removes last element
numbers.unshift(0); // Adds 0 at the beginning
numbers.shift(); // Removes the first element

let squaredNumbers = numbers.map(num => num * num);
console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

TypeScript Union Types

Union types in TypeScript allow a variable to hold values of multiple types, providing flexibility while maintaining type safety.

📌 Defining Union Types

You can define a union type using the `|` (pipe) symbol:

```
let value: string | number;
value = "Hello"; // ✅ Allowed
value = 42;      // ✅ Allowed
// value = true; // ❌ Error: Type 'boolean' is not assignable to type 'string | number'
```

📌 Union Types with Functions

```
function printId(id: string | number) {
    console.log("ID:", id);
}

printId(101);     // ✅ Allowed
printId("A123"); // ✅ Allowed
// printId(true); // ❌ Error
```

📌 Union Types with Arrays

You can define an array with mixed types:

```
let mixedArray: (string | number)[] = [1, "two", 3, "four"];
```

📌 Narrowing Union Types

To safely handle different types, use **type narrowing**:

```
function getLength(value: string | number): number {
    if (typeof value === "string") {
        return value.length; // Allowed because TypeScript knows it's a string
    }
    return value.toString().length; // Convert number to string to get length
}
```

```
console.log(getLength("Anik")); // Output: 4
console.log(getLength(12345));  // Output: 5
```

📌 Union Types with Objects

You can use union types with objects to define multiple possible structures:

```
type Dog = {
    bark: () => void;
};

type Cat = {
    meow: () => void;
};

let pet: Dog | Cat;

pet = { bark: () => console.log("Woof!") }; // ✅ Allowed
pet = { meow: () => console.log("Meow!") }; // ✅ Allowed
// pet = { fly: () => console.log("Flap!") }; // ❌ Error
```

TypeScript Tuples

Tuples in TypeScript allow storing multiple values with **fixed types and order** in a single array-like structure.

📌 Defining Tuples

A tuple is defined by specifying types for each position in the array.

```
let userInfo: [string, number] = ["Anik", 25];
console.log(userInfo[0]); // Output: "Anik"
console.log(userInfo[1]); // Output: 25
```

📌 Accessing Tuple Elements

Tuple elements can be accessed using **indexing** like an array.

```
let person: [string, number, boolean] = ["John", 30, true];
console.log(person[0]); // Output: "John"
```

```
console.log(person[1]); // Output: 30
console.log(person[2]); // Output: true
```

📌 Modifying Tuples

Tuples allow modifying elements but must maintain the correct type.

```
let employee: [string, number] = ["Alice", 101];
employee[1] = 102; // ☑ Allowed
// employee[0] = 100; // ✖ Error: Type 'number' is not assignable to type
'string'
```

📌 Tuple with Optional Elements

Tuples can have optional elements using `?`.

```
let student: [string, number, string?] = ["Bob", 20];
student = ["Bob", 20, "A+"]; // ☑ Allowed
```

📌 Tuple with Rest Parameters

Tuples support rest parameters for **variable-length elements**.

```
let scores: [string, ...number[]] = ["Anik", 90, 85, 88];
console.log(scores); // Output: ["Anik", 90, 85, 88]
```

📌 Tuple Methods

Tuples can use array methods, but type safety is enforced.

```
let data: [string, number] = ["Alice", 101];
data.push("Developer"); // ☑ Allowed, but not recommended
console.log(data); // Output: ["Alice", 101, "Developer"]
```

📌 Exception in Tuples

Tuples enforce strict type and order, but methods like `push` allow adding elements, which can violate tuple constraints.

```
//---Exception---
type nodeUserA= [number ,string];

const nodeA: nodeUserA=[1,"Block-A"];
nodeA.push("Hola"); //=>This push operation shouldn't be done, it's also violating the sequence.
nodeA.push(1);  //=>This push operation shouldn't be done, it's also violating the sequence.
console.log(nodeA);
```

☑ Solution: Making the Tuple Immutable

To prevent accidental modifications, use `readonly`.

```
type nodeUserB=readonly [number ,string];
const nodeB: nodeUserB=[1,"Block-A"];
// nodeB.push("Hola"); //=>Throws error
// nodeB.push(1);  //=>Throws error
console.log(nodeB);
```

🚀 Conclusion

Tuples in TypeScript provide a **structured** way to handle multiple values with different types in a single variable. They improve **type safety** while keeping flexibility. Let me know if you need further improvements! 🚀

TypeScript Enums

Enums in TypeScript allow defining a **set of named constants**, making code more readable and maintainable.

📌 Defining an Enum

Enums can be defined using the `enum` keyword.

```
enum Role {
    Admin,
    User,
    Guest
}
console.log(Role.Admin); // Output: 0
console.log(Role.User);  // Output: 1
console.log(Role.Guest); // Output: 2
```

By default, TypeScript assigns numeric values starting from `0`, incrementing by `1`.

📌 Assigning Custom Values

Custom values can be assigned to enum members.

```
enum Status {
    Success = 200,
    NotFound = 404,
    ServerError = 500
}
console.log(Status.Success); // Output: 200
console.log(Status.NotFound); // Output: 404
```

📌 String Enums

Enums can also have **string values**.

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT"
}
console.log(Direction.Up); // Output: "UP"
```

📌 Heterogeneous Enums

Enums can have both **numeric and string values** (not recommended).

```
enum Mix {
    Yes = "YES",
    No = 0
}
console.log(Mix.Yes); // Output: "YES"
console.log(Mix.No);  // Output: 0
```

📌 Reverse Mapping

TypeScript allows reverse mapping for **numeric enums**.

```
enum Color {
    Red = 1,
    Green,
    Blue
}
console.log(Color[2]); // Output: "Green"
```

📌 Using `const enum`

A `const enum` removes runtime overhead by inlining values directly.

```
const enum Size {
    Small = 1,
    Medium = 2,
    Large = 3
}
console.log(Size.Small); // Output: 1
```

🚀 Conclusion

Enums in TypeScript improve readability by giving meaningful names to constant values. They can be numeric, string-based, or a mix of both. Using `const enum` optimizes performance by eliminating extra code at runtime. 🚀

TypeScript Interfaces

Interfaces in TypeScript define the **structure of an object**, enforcing **type safety** while allowing flexibility.

📌 Defining an Interface

An interface specifies the expected structure of an object.

```
interface User {
    name: string;
    age: number;
    email?: string; // Optional property
}

let user1: User = {
    name: "Anik",
    age: 25,
```

```
};
console.log(user1);
```

📌 Readonly Properties

Use `readonly` to prevent modification of certain properties.

```
interface Product {
    readonly id: number;
    name: string;
}

let item: Product = { id: 101, name: "Laptop" };
// item.id = 102; // ❌ Error: Cannot modify readonly property
```

📌 Extending Interfaces

Interfaces can inherit properties from other interfaces.

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

interface Employee extends Person {
    employeeId: number;
}

let emp: Employee = { name: "John", age: 30, employeeId: 1234 };
console.log(emp);
```

📌 Interface Reopening

Unlike types, interfaces can be reopened and extended multiple times.

```
interface Car {
    brand: string;
}

interface Car {
    model: string;
}
```

```
let myCar: Car = { brand: "Tesla", model: "Model S" };
console.log(myCar);
```

📌 Implementing Interfaces in Classes

Classes can **implement** an interface to ensure they follow a specific structure.

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

class Dog implements Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    makeSound() {
        console.log("Woof! Woof!");
    }
}

const myDog = new Dog("Buddy");
myDog.makeSound();
```

📌 Function Interfaces

Interfaces can define function structures.

```
interface MathOperation {
    (a: number, b: number): number;
}

const add: MathOperation = (x, y) => x + y;
console.log(add(5, 3)); // Output: 8
```

📌 Index Signatures

Index signatures allow defining **dynamic object properties**.

```
interface UserDictionary {
    [key: string]: string;
}
```

```
let users: UserDictionary = {
    "user1": "Anik",
    "user2": "John",
};
console.log(users["user1"]);
```

📌 Differences Between Types and Interfaces

Feature	Interface	Type
Object Shapes	✅ Preferred	✅ Possible
Reopening	✅ Yes	❌ No
Declaration Merging	✅ Yes	❌ No
Function Signatures	✅ Yes	✅ Yes
Union & Intersection	❌ No	✅ Yes

```
// Using an interface
interface Person {
    name: string;
    age: number;
}

// Using a type
type Employee = {
    name: string;
    age: number;
    position: string;
};
```

🚀 Conclusion

Interfaces in TypeScript enforce **structure and type safety** while allowing flexibility. They work with **objects, classes, and functions** to enhance maintainability. Additionally, interfaces and types serve different purposes, and understanding when to use each can improve code clarity and reusability. Let me know if you need further improvements! 🚀

TypeScript Classes

TypeScript provides **class-based object-oriented programming** with **strong typing**. Classes help in structuring code, encapsulating data, and defining behavior.

📌 Defining a Class

```
class User {
    name: string;
    age: number;

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

const user1 = new User("Anik", 25);
console.log(user1);
```

📌 Access Modifiers

Access modifiers control the visibility of properties and methods:

- `public`: Accessible everywhere.
- `private`: Accessible only within the class.
- `protected`: Accessible within the class and subclasses.

```
class Employee {
    public name: string;
    private salary: number;
    protected department: string;

    constructor(name: string, salary: number, department: string) {
        this.name = name;
        this.salary = salary;
        this.department = department;
    }
}
```

📌 Readonly Properties

The `readonly` keyword ensures that a property cannot be modified after initialization.

```
class Product {
    readonly id: number = Math.random();
    name: string;

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

```
    }
}
```

📌 Getters and Setters

Encapsulate property access using **getters and setters**.

```
class Account {
    private _balance: number = 1000;

    get balance(): number {
        return this._balance;
    }

    set balance(amount: number) {
        if (amount > 0) {
            this._balance = amount;
        }
    }
}

const acc = new Account();
console.log(acc.balance); // 1000
acc.balance = 1500;
console.log(acc.balance); // 1500
```

📌 Inheritance

A class can **inherit properties and methods** from another class using `extends`.

```
class Animal {
    constructor(public name: string) {}
    makeSound() {
        console.log("Some sound...");
    }
}

class Dog extends Animal {
    makeSound() {
        console.log("Woof! Woof!");
    }
}

const myDog = new Dog("Buddy");
myDog.makeSound(); // Woof! Woof!
```

📌 Abstract Classes

An **abstract class** cannot be instantiated directly and is used as a blueprint for derived classes.

```
abstract class Vehicle {
    constructor(public brand: string) {}
    abstract drive(): void;
}

class Car extends Vehicle {
    drive() {
        console.log("Driving a car...");
    }
}

const myCar = new Car("Tesla");
myCar.drive();
```

📌 Implementing Interfaces

Classes can implement **interfaces** to enforce structure.

```
interface Shape {
    area(): number;
}

class Circle implements Shape {
    constructor(public radius: number) {}
    area(): number {
        return Math.PI * this.radius * this.radius;
    }
}
```

📌 Class Example with Readonly and Private Properties

```
enum Role {
    ADMIN,
    CLIENT
}

class User {
    readonly id: number = Math.random();
    constructor(
        public username: string,
        public email: string,
        public city: string,
```

```
        private role: Role = Role.CLIENT
    ) {}
}
```

🚀 Conclusion

TypeScript classes provide a powerful way to structure code using **OOP principles**. With features like **access modifiers, inheritance, and interfaces**, TypeScript ensures robust and maintainable applications.

TypeScript Abstract Classes

Abstract classes in TypeScript provide a blueprint for other classes. They cannot be instantiated directly and must be extended by subclasses.

📌 Defining an Abstract Class

An abstract class can include both **implemented methods** and **abstract methods** (methods without a body that must be implemented by subclasses).

```typescript
abstract class Animal {
    constructor(public name: string) {}
    abstract makeSound(): void; // Abstract method

    move(): void {
        console.log(`${this.name} is moving...`);
    }
}
```

📌 Extending an Abstract Class

A subclass must implement all abstract methods of the parent class.

```typescript
class Dog extends Animal {
    makeSound(): void {
        console.log("Woof! Woof!");
    }
}

const myDog = new Dog("Buddy");
myDog.makeSound(); // Output: Woof! Woof!
myDog.move(); // Output: Buddy is moving...
```

📌 Abstract Properties

Abstract classes can have abstract properties that must be defined in subclasses.

```
abstract class Vehicle {
    abstract speed: number;
    abstract accelerate(): void;
}

class Car extends Vehicle {
    speed = 120;
    accelerate() {
        console.log(`The car accelerates at ${this.speed} km/h`);
    }
}
```

📌 Abstract Class with Constructor

Abstract classes can also have constructors to enforce initialization.

```
abstract class Employee {
    constructor(public name: string, public role: string) {}
    abstract work(): void;
}

class Developer extends Employee {
    work() {
        console.log(`${this.name} is coding as a ${this.role}`);
    }
}

const dev = new Developer("Anik", "Frontend Developer");
dev.work(); // Output: Anik is coding as a Frontend Developer
```

📌 Difference Between Abstract Classes and Interfaces

Feature	Abstract Class	Interface
Can have method implementations	☑ Yes	✖ No
Can have abstract methods	☑ Yes	☑ Yes
Can be instantiated	✖ No	✖ No
Can have constructors	☑ Yes	✖ No

Feature	Abstract Class	Interface
Supports multiple inheritance	✖ No	☑ Yes
Used for	Shared behavior and structure	Defining a contract

🚀 Key Takeaways

- **Abstract classes cannot be instantiated directly.**
- **They define a common structure for subclasses.**
- **Abstract methods must be implemented in derived classes.**
- **They can include concrete methods and properties.**
- **Interfaces define a contract, while abstract classes provide shared behavior.**

TypeScript Generics

Generics in TypeScript allow us to create reusable and flexible components while maintaining type safety. They enable us to work with various data types without sacrificing type inference.

📌 Defining Generics

Generics are defined using angle brackets `<T>` after function, class, or interface names.

```
function identity<T>(value: T): T {
    return value;
}

console.log(identity<number>(42)); // Output: 42
console.log(identity<string>("Hello")); // Output: Hello
```

📌 Generic Functions

Functions can accept generic types to make them more reusable.

```
function merge<T, U>(obj1: T, obj2: U): T & U {
    return { ...obj1, ...obj2 };
}

const mergedObj = merge({ name: "Anik" }, { age: 20 });
console.log(mergedObj); // Output: { name: "Anik", age: 20 }
```

📌 Generic Interfaces

Interfaces can also use generics to provide flexibility in type definitions.

```
interface Box<T> {
    content: T;
}

const numberBox: Box<number> = { content: 100 };
const stringBox: Box<string> = { content: "TypeScript" };
```

📌 Generic Classes

Classes can use generics to handle different types dynamically.

```
class DataStorage<T> {
    private data: T[] = [];

    addItem(item: T) {
        this.data.push(item);
    }

    getAllItems(): T[] {
        return this.data;
    }
}

const textStorage = new DataStorage<string>();
textStorage.addItem("Hello");
console.log(textStorage.getAllItems()); // Output: ["Hello"]
```

📌 Generic Constraints

We can restrict generics to specific types using `extends`.

```
function getLength<T extends { length: number }>(item: T): number {
    return item.length;
}

console.log(getLength("Hello")); // Output: 5
console.log(getLength([1, 2, 3])); // Output: 3
```

📌 Using Comma in Generics

Generics can accept multiple type parameters separated by commas.

```
function pair<T, U>(first: T, second: U): [T, U] {
    return [first, second];
}

const result = pair<number, string>(10, "Anik");
console.log(result); // Output: [10, "Anik"]
```

🚀 Key Takeaways

- **Generics provide type safety while maintaining flexibility.**
- **They allow the creation of reusable functions, classes, and interfaces.**
- **Constraints help limit the types generics can accept.**
- **Multiple type parameters can be defined using a comma.**

Generics are a powerful feature in TypeScript, making code more reusable and scalable. Let me know if you need further clarification! 🚀

TypeScript Type Narrowing

Type narrowing in TypeScript refers to refining the type of a variable within a specific block of code based on runtime checks. It ensures type safety and allows TypeScript to infer more specific types.

📌 Type Guards

Type guards are used to narrow down types using condition checks.

`typeof` Type Guard

The `typeof` operator helps check primitive types.

```
function printId(id: string | number) {
    if (typeof id === "string") {
        console.log("ID is a string: ", id.toUpperCase());
    } else {
        console.log("ID is a number: ", id.toFixed(2));
    }
}
```

`instanceof` Type Guard

The `instanceof` operator helps check if an object belongs to a particular class.

```
class Car {
    drive() {
        console.log("Driving a car...");
    }
}

class Bike {
    ride() {
        console.log("Riding a bike...");
    }
}

function useVehicle(vehicle: Car | Bike) {
    if (vehicle instanceof Car) {
        vehicle.drive();
    } else {
        vehicle.ride();
    }
}
```

`in` Operator Type Guard

The `in` operator checks if a property exists in an object.

```
interface Dog {
    bark: () => void;
}

interface Cat {
    meow: () => void;
}

function makeSound(animal: Dog | Cat) {
    if ("bark" in animal) {
        animal.bark();
    } else {
        animal.meow();
    }
}
```

📌 Discriminated Unions

Using a common property (discriminator) to differentiate between object types.

```typescript
interface Circle {
    kind: "circle";
    radius: number;
}

interface Square {
    kind: "square";
    sideLength: number;
}

function getArea(shape: Circle | Square) {
    if (shape.kind === "circle") {
        return Math.PI * shape.radius ** 2;
    } else {
        return shape.sideLength ** 2;
    }
}
```

📌 User-Defined Type Guards

User-defined type guards use type predicates (`animal is Type`) to explicitly tell TypeScript which type an object belongs to.

```typescript
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function isFish(animal: Fish | Bird): animal is Fish {
    return (animal as Fish).swim !== undefined;
}

function getFood(animal: Fish | Bird) {
    if (isFish(animal)) {
        animal; // Now TypeScript treats `animal` as Fish
        return "Fish Food";
    } else {
        animal; // Now TypeScript treats `animal` as Bird
        return "Bird Food";
    }
}
```

Explanation

- The `isFish` function acts as a **user-defined type guard**.
- It explicitly tells TypeScript that if `isFish(animal)` returns `true`, then `animal` is of type `Fish`.
- Inside `getFood`, TypeScript correctly narrows `animal` to either `Fish` or `Bird` based on the guard.

📌 Type Assertions (Use with Caution)

Type assertions manually specify a type when TypeScript cannot infer it correctly.

```
let input = document.getElementById("username") as HTMLInputElement;
input.value = "Anik";
```

🚀 Key Takeaways

- **Type narrowing refines a variable's type for better type safety.**
- **Use `typeof`, `instanceof`, and `in` for type guards.**
- **Discriminated unions improve type differentiation.**
- **User-defined type guards provide custom logic for narrowing types.**
- **Type assertions should be used cautiously.**

Mastering type narrowing helps write cleaner and more robust TypeScript code! 🚀