TypeScript @Sagar Sahu

TypeScript is a programming language that adds extra features to JavaScript. It helps you write more reliable code by letting you define what type of data (like numbers, strings, or objects) each part of your code should use.

Think of TypeScript as a tool that helps you catch mistakes in your code before your program runs. It's like having a checklist that ensures everything is correct before you start using it.

▼ Basic Types in TypeScript

TypeScript extends JavaScript by adding types to variables, enabling better error detection and code clarity. Basic types are the foundation of TypeScript, and they represent the simplest forms of data you'll work with.

List of Basic Types

1. `string`: Represents text.
2. `number`: Represents numeric values.
3. `boolean`: Represents true/false values.
4. `array`: Represents a collection of values.
5. `tuple`: Represents a fixed-size array with specific types at each index.
6. `enum`: Represents a set of named constants.
7. `any`: Represents any type (avoids type checking).
8. `unknown`: Represents a safer version of `any`.
9. `void`: Represents no value or absence of return.
10. `null` **and** `undefined`: Represent empty or uninitialized values.
11. `never`: Represents values that never occur.

In-Depth Explanation of Each Type

1. `string`

Used to store textual data.

Example:

```
let firstName: string = "Alice";
let greeting: string = `Hello, ${firstName}`; // Using
template literals
console.log(greeting); // Output: Hello, Alice
```

---

2. `number`

Used for all numbers, including integers and floating-point values.

Example:

```
let age: number = 24;
let pi: number = 3.14;
console.log(age, pi); // Output: 24 3.14
```

---

3. `boolean`

Used for true/false values.

Example:

```
let isLoggedIn: boolean = true;
console.log(isLoggedIn); // Output: true
```

---

4. `array`

Used to store a list of values. Can be typed with a specific type of elements.

Example:

Using generic type (`Array<Type>`):

```
let numbers: Array<number> = [1, 2, 3];
console.log(numbers); // Output: [1, 2, 3]
```

Using square brackets (`Type[]`):

```
let names: string[] = ["Alice", "Bob"];
console.log(names); // Output: ["Alice", "Bob"]
```

---

5. `tuple`

An array with a fixed number of elements, where each element has a specific type.

Example:

```
let user: [string, number] = ["Alice", 24];
console.log(user); // Output: ["Alice", 24]
```

---

6. `enum`

Used for defining a set of named constants.

Example:

```
enum Color {
    Red,
    Green,
    Blue
}

let myColor: Color = Color.Green;
console.log(myColor); // Output: 1 (index of Green in t
he enum)
```

---

7. `any`

Allows a variable to hold any type. Use cautiously, as it disables type checking.

Example:

```
let value: any = "Hello";
value = 42; // Allowed
console.log(value); // Output: 42
```

---

8. `unknown`

Similar to `any`, but safer as it requires type checking before usage.

Example:

```
let value: unknown = "Hello";

// value.toUpperCase(); // Error: Object is of type 'un
known'

if (typeof value === "string") {
    console.log(value.toUpperCase()); // Output: HELLO
}
```

---

9. `void`

Represents the absence of a value, typically used for functions that don't return anything.

Example:

```
function logMessage(message: string): void {
    console.log(message);
}

logMessage("Hello, TypeScript!"); // Output: Hello, Typ
eScript!
```

---

10. `null` and `undefined`

- `null`: Represents an intentionally empty value.
- `undefined`: Represents an uninitialized value.

Example:

```
let value: null = null;
let anotherValue: undefined = undefined;
console.log(value, anotherValue); // Output: null undef
ined
```

---

11. `never`

Represents values that never occur, such as functions that always throw an error or never return.

Example:

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

Key Points to Remember

1. **Type Checking**: Basic types ensure that the wrong type of value is caught during development.
2. **Strict Mode**: Enabling strict type checks in your `tsconfig.json` ensures safer, more robust code.
3. **Use Explicit Types**: Always declare types explicitly for better readability and fewer errors.

▼ 1. Static Typing **(Type Annotations)**

Static Typing in TypeScript

Static Typing is a key feature in TypeScript that allows you to define the type of a variable, function parameter, or return value at the time of writing your code. This is different from JavaScript, where types are determined at runtime. With static typing, TypeScript checks your code for type errors before it even runs, helping you catch mistakes early.

1. Type Annotations

Type annotations in TypeScript are a way to explicitly declare the type of a variable, function parameter, or return value. This means you tell TypeScript what kind of data a variable should hold.

Example of Type Annotations

```
let message: string = "Hello, TypeScript!"; // 'messag
e' must always be a string
let count: number = 42;                     // 'count' m
ust always be a number
let isDone: boolean = true;                 // 'isDone'
must always be a boolean
```

Remember:

- `string`, `number`, and `boolean` are basic types.
- If you try to assign a value of a different type to these variables, TypeScript will show an error.

Why is this important?

- **Error Prevention**: Static typing helps you avoid common errors, like accidentally assigning a string to a number variable.
- **Code Clarity**: When you or someone else reads your code, the types make it clear what kind of data to expect, making the code easier to understand and maintain.

Type Annotations in Functions

You can also use type annotations in functions to specify the types of the parameters and the return value.

```
function add(a: number, b: number): number {
    return a + b;
}

console.log(add(5, 3));  // Output: 8
```

Key Points to Remember:

- `a: number` and `b: number` specify that `a` and `b` must be numbers.
- `: number` after the parentheses indicates that the function returns a number.

If you try to call the `add` function with anything other than numbers, TypeScript will give you an error.

Type Inference

TypeScript can automatically infer types based on the value assigned to a variable, so you don’t always need to use type annotations.

```
let name = "Sagar";  // TypeScript infers that 'name' i
s a string
let age = 24;        // TypeScript infers that 'age' is
a number
```

Type inference is useful, but explicitly using type annotations can make your code more understandable and error-proof.

How to Remember Static Typing:

1. **Static = Set in Stone**: Once you define a type, it doesn't change. Like setting something in stone, it stays the same.
2. **Type Annotations = Labels**: Think of type annotations as labels you stick on variables, telling everyone (and TypeScript) exactly what's inside.
3. **Error Prevention**: Static typing acts like a safety net, catching mistakes before they cause problems.
4. **Code Clarity**: With types clearly marked, anyone can quickly understand what your code is doing.

By using static typing and type annotations in TypeScript, you write safer, more reliable code that's easier to read and maintain.

▼ 2.Interfaces in TypeScript

Interfaces in TypeScript are like blueprints that define the structure of an object. They allow you to specify what properties and methods an object should have, along with their types. Interfaces ensure that objects follow a certain structure, helping to catch errors and making your code more predictable.

1. Basic Structure of an Interface

An interface is defined using the `interface` keyword, followed by the name of the interface. Inside the interface, you declare properties and their types.

Example of a Simple Interface

```
interface User {
    name: string;
    age: number;
    isActive: boolean;
}

let user1: User = {
    name: "Sagar",
    age: 24,
```

```
    isActive: true,
};

console.log(user1); // Output: { name: "Sagar", age: 2
4, isActive: true }
```

Remember:

- The `User` interface defines that any object of type `User` must have `name`, `age`, and `isActive` properties with specific types (`string`, `number`, `boolean`).
- If you try to create an object without following this structure, TypeScript will throw an error.

2. Optional Properties

In an interface, you can define optional properties by using a `?` after the property name. Optional properties don't have to be included in the object.

Example with Optional Properties

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

let user2: User = {
    name: "Amit",
    age: 25,
    isActive: false,
    // email is optional, so we can skip it
};

console.log(user2); // Output: { name: "Amit", age: 25,
isActive: false }
```

Key Points:

- `email` is optional, so `user2` can be created without it.
- This flexibility is useful when not every object needs every property.

3. Read-only Properties

You can use the `readonly` modifier to make properties in an interface immutable after they are set.

Example with Read-Only Properties

```
interface User {
    readonly id: number;
    name: string;
    age: number;
}

let user3: User = {
    id: 101,
    name: "Kiran",
    age: 26,
};

// user3.id = 102; // Error: Cannot assign to 'id' beca
use it is a read-only property.
```

Key Points:

- `id` is marked as `readonly`, meaning once it's set, it can't be changed.
- This ensures that critical properties remain constant, adding safety to your code.

4. Interface for Functions

Interfaces can also be used to define the structure of functions, including the parameters and return type.

Example of an Interface for a Function

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

let sayHello: Greet = function(name: string) {
    return `Hello, ${name}!`;
};
```

```
console.log(sayHello("Sagar")); // Output: "Hello, Saga
r!"
```

Remember:

- The `Greet` interface defines that any function of this type must take a `string` as a parameter and return a `string`.

5. Extending Interfaces

You can create new interfaces by extending existing ones, which allows you to build on previous structures without rewriting them.

Example of Extending an Interface

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

interface Employee extends Person {
    employeeId: number;
}

let employee1: Employee = {
    name: "John",
    age: 30,
    employeeId: 1234,
};

console.log(employee1); // Output: { name: "John", age:
30, employeeId: 1234 }
```

Key Points:

- `Employee` extends `Person`, so it inherits `name` and `age` while adding `employeeId`.
- This helps in reusing code and keeping your interfaces DRY (Don’t Repeat Yourself).

How to Remember Interfaces:

1. **Blueprint**: Think of an interface as a blueprint or a contract that objects must follow.
2. **Optional Properties**: `?` means the property is optional, like leaving a field blank in a form.
3. **Read-Only**: `readonly` is like a lock on a property, preventing changes after it's set.
4. **Functions**: Interfaces can describe the structure of functions too, not just objects.
5. **Inheritance**: Interfaces can be extended, like building new floors on an existing building.

Interfaces help you define clear, consistent structures for your objects, leading to more maintainable and error-free code.

▼ 3.Union Types in TypeScript

Union Types in TypeScript allow a variable to hold more than one type of value. This is useful when a variable can be one of several types, giving you flexibility while still enforcing type safety.

1. Basic Syntax

To use a union type, you use the pipe symbol (`|`) to separate the different types a variable can hold.

Example of Union Types

```
let value: string | number;

value = "Hello";  // OK
value = 42;       // OK
// value = true;  // Error: Type 'boolean' is not assig
nable to type 'string | number'.
```

Remember:

- `value` can be either a `string` or a `number`, but not a `boolean`.

2. Using Union Types in Functions

Union types are also useful for function parameters and return values, allowing functions to accept or return multiple types.

Example of Union Types in Functions

```
function printId(id: number | string): void {
    console.log(`Your ID is: ${id}`);
}

printId(123);      // Output: Your ID is: 123
printId("ABC123"); // Output: Your ID is: ABC123
```

Key Points:

- `printId` can accept either a `number` or a `string` as its argument.

3. Type Narrowing

When you use union types, TypeScript often needs to determine which type is being used at a specific point in the code. This process is called **type narrowing**. TypeScript uses conditions to figure out which type is currently being used.

Example of Type Narrowing

```
function printLength(value: string | number): void {
    if (typeof value === "string") {
        console.log(`String length: ${value.length}`);
    } else {
        console.log(`Number value: ${value}`);
    }
}

printLength("Hello"); // Output: String length: 5
printLength(123);     // Output: Number value: 123
```

How it works:

- The `typeof` operator is used to check the type of `value`.
- If `value` is a string, `value.length` is used. If it's a number, just print the number.

4. Union Types with Objects

Union types can also be used with objects, allowing different object shapes.

Example with Object Union Types

```
interface Cat {
    type: "cat";
    meow: () => void;
}

interface Dog {
    type: "dog";
    bark: () => void;
}

type Pet = Cat | Dog;

function interactWithPet(pet: Pet): void {
    if (pet.type === "cat") {
        pet.meow();
    } else {
        pet.bark();
    }
}

const myCat: Cat = { type: "cat", meow: () => console.l
og("Meow!") };
const myDog: Dog = { type: "dog", bark: () => console.l
og("Woof!") };

interactWithPet(myCat); // Output: Meow!
interactWithPet(myDog); // Output: Woof!
```

Key Points:

- `Pet` can be either a `Cat` or a `Dog`.
- Type narrowing is used to handle the different methods (`meow` or `bark`) based on the type of `pet`.

How to Remember Union Types:

1. **Pipe Symbol (`|`)**: Think of it as saying “or.” You can have `string` **or** `number`.
2. **Flexibility**: Union types give you flexibility by allowing multiple possible types for a variable.

3. **Type Narrowing**: TypeScript helps you figure out which type you're dealing with at any point in your code.

Union types are powerful tools in TypeScript that help you handle variables and functions that can work with multiple types, improving the flexibility and safety of your code.

▼ 4.Generics in TypeScript

Generics in TypeScript are a way to create reusable components or functions that work with different data types. They allow you to define a placeholder for a type, which you can then specify later when you use the component or function. This makes your code more flexible and type-safe.

1. Basics of Generics

Generics use a placeholder `<T>` (or any letter) to represent a type. You can then specify the actual type when you use the generic component or function.

Example of a Generic Function

Here's a simple generic function that returns the same type that it receives.

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

console.log(identity("Hello")); // Output: Hello
console.log(identity(42));      // Output: 42
```

Key Points:

- `<T>` is a placeholder for any type.
- The function `identity` works with any type (string, number, etc.) and returns that same type.

2. Generics with Classes

Generics can also be used with classes to create data structures that can work with different types.

Example of a Generic Class

```
class Box<T> {
    private value: T;
```

```
    constructor(value: T) {
        this.value = value;
    }

    getValue(): T {
        return this.value;
    }
}

const stringBox = new Box<string>("A String");
console.log(stringBox.getValue()); // Output: A String

const numberBox = new Box<number>(123);
console.log(numberBox.getValue()); // Output: 123
```

Key Points:

- `Box<T>` can hold a value of any type.
- When creating a `Box`, you specify the type of value it will hold (`string` or `number` in this case).

3. Generics with Interfaces

Generics can be used with interfaces to define contracts for various types.

Example of a Generic Interface

```
interface Result<T> {
    data: T;
    error: string | null;
}

const successResult: Result<string> = {
    data: "Everything is fine",
    error: null,
};

const failureResult: Result<number> = {
    data: 0,
    error: "An error occurred",
};
```

```
console.log(successResult); // Output: { data: "Everyth
ing is fine", error: null }
console.log(failureResult); // Output: { data: 0, erro
r: "An error occurred" }
```

Key Points:

- `Result<T>` can describe the result of an operation with any type of data (`string`, `number`, etc.).

4. Constraints on Generics

You can restrict the types that can be used with a generic by adding constraints. This ensures that the generic type adheres to a certain shape or behavior.

Example of Generics with Constraints

```
interface Lengthwise {
    length: number;
}

function logLength<T extends Lengthwise>(value: T): voi
d {
    console.log(`Length: ${value.length}`);
}

logLength("Hello");      // Output: Length: 5
logLength([1, 2, 3]);   // Output: Length: 3
// logLength(42);        // Error: Argument of type 'nu
mber' is not assignable to parameter of type 'Lengthwis
e'.
```

Key Points:

- `T extends Lengthwise` ensures that `T` has a `length` property.
- This allows `logLength` to work only with types that have a `length` property.

How to Remember Generics:

1. **Placeholders for Types**: Generics are like placeholders (`<T>`) for types. They let you write code that can handle many types, not just one.
2. **Flexibility and Reusability**: Generics make your functions and classes flexible and reusable with different types.
3. **Constraints**: You can limit what types can be used with generics to ensure they meet certain requirements (like having a `length` property).

Generics in TypeScript help you create flexible and reusable components while maintaining type safety, making your code more robust and adaptable.

▼ 5. **Type Inference**:

Type Inference in TypeScript

Type Inference is a feature in TypeScript where the compiler automatically determines the type of a variable based on its value. This means you don't always need to explicitly specify types because TypeScript can "infer" them from the context.

1. How Type Inference Works

When you declare a variable and assign a value to it, TypeScript will infer the type of the variable from the assigned value.

Example of Type Inference

```
let message = "Hello, TypeScript!"; // TypeScript infer
s 'string'
let count = 42;                     // TypeScript infer
s 'number'
let isValid = true;                 // TypeScript infer
s 'boolean'

// message = 123;  // Error: Type 'number' is not assig
nable to type 'string'
```

Key Points:

- `message` is inferred as `string` because of the initial value `"Hello, TypeScript!"`.

- You cannot assign a value of a different type to the variable after initialization.

2. Type Inference in Functions

For functions, TypeScript can infer the return type based on the function's implementation.

Example of Type Inference in Functions

```
function add(a: number, b: number) {
    return a + b; // TypeScript infers the return type
as 'number'
}

const result = add(10, 20); // TypeScript knows 'resul
t' is a number
console.log(result);        // Output: 30
```

Key Points:

- TypeScript infers the return type of the `add` function as `number` from the `a + b` operation.
- Explicitly declaring the return type is optional when inference works correctly.

3. Contextual Typing

TypeScript can infer the type of a variable or function based on its context, such as in event handlers or callbacks.

Example of Contextual Typing

```
document.addEventListener("click", (event) => {
    console.log(event.clientX, event.clientY); // 'even
t' is inferred as MouseEvent
});
```

Key Points:

- The `event` parameter is automatically inferred as a `MouseEvent` because the callback is used in a `click` event handler.

4. Best Practices with Type Inference

While type inference reduces the need for explicit type annotations, there are situations where you should still provide explicit types for clarity, especially for function parameters and return types.

Example of Explicit Types vs. Inferred Types

```
// Inferred type
let name = "Alice"; // TypeScript infers 'string'

// Explicit type (useful for clarity or complex scenari
os)
let age: number = 25; // Explicitly declares 'number' t
ype
```

When to Use Explicit Types:

- When the inferred type is not obvious.
- When the code's readability or intent can benefit from explicit annotation.

How to Remember Type Inference:

1. **Saves Effort**: TypeScript automatically assigns types based on values, saving you from writing redundant type annotations.
2. **Automatic but Smart**: TypeScript will infer types wherever it can, but you can always explicitly declare them if needed.
3. **Clarity is Key**: Rely on type inference for simple cases, but use explicit types for complex ones or when the inferred type isn’t clear.

Type inference is a powerful feature in TypeScript that simplifies coding while maintaining type safety.

▼ 6. Optional **Properties/Parameters**:

In TypeScript, you can define properties and parameters as **optional**. This means the value is not required, and the property or parameter may or may not be provided.

Optional Properties in Interfaces

When defining an interface, you can make a property optional by adding a `?` after its name. If the property is not provided, TypeScript will not throw an error.

Example:

```
interface User {
    name: string;
    age?: number; // 'age' is optional
}

const user1: User = { name: "Alice" };       // Valid:
'age' is not provided
const user2: User = { name: "Bob", age: 30 }; // Valid:
'age' is provided

console.log(user1); // Output: { name: "Alice" }
console.log(user2); // Output: { name: "Bob", age: 30 }
```

Key Points:

- The `?` makes the `age` property optional.
- If `age` is not provided, TypeScript does not raise an error.

Optional Parameters in Functions

When defining a function, you can make a parameter optional using the same `?`. Optional parameters must always come **after required parameters**.

Example:

```
function greet(name: string, age?: number): string {
    if (age) {
        return `Hello, ${name}. You are ${age} years ol
d.`;
    }
    return `Hello, ${name}.`;
}

console.log(greet("Alice"));       // Output: Hello, Al
ice.
console.log(greet("Bob", 30));     // Output: Hello, Bo
b. You are 30 years old.
```

Key Points:

- `age` is an optional parameter. It can be provided or omitted.
- The function handles both cases (`age` provided or not) correctly.

Using Optional Properties with Default Values

Optional properties do not have default values unless explicitly assigned.

Example:

```
interface Config {
    timeout?: number; // Optional, default behavior dep
ends on usage
}

const defaultConfig: Config = {};
const customConfig: Config = { timeout: 5000 };

console.log(defaultConfig.timeout); // Output: undefine
d
console.log(customConfig.timeout);  // Output: 5000
```

Remember This:

1. **Syntax**: Use `?` to make properties or parameters optional.
2. **Optional Properties**:
    - Allow objects to omit certain fields.
3. **Optional Parameters**:
    - Allow functions to be called with fewer arguments.
4. **Safe Usage**: Always handle the possibility of `undefined` when accessing optional values.

By using optional properties and parameters, you can make your code more flexible and easier to use, while still maintaining type safety.

▼ 7. **Enums**

Enums are a special feature in TypeScript that allow you to define a set of named constants. They make code more readable and maintainable by grouping related values together.

Types of Enums in TypeScript

1. Numeric Enums

Default behavior: Members are automatically assigned numeric values starting from `0`.

Example:

```
enum Status {
    Pending,   // 0
    InProgress, // 1
    Completed   // 2
}

console.log(Status.Pending);     // Output: 0
console.log(Status.InProgress);  // Output: 1
console.log(Status.Completed);   // Output: 2
```

Key Points:

- The first member (`Pending`) gets `0` by default, and the rest increment by `1`.
- You can access enum values using both names (`Status.Pending`) and numbers (`Status[0]`).

---

2. Custom Numeric Enums

You can assign custom numbers to enum members. Subsequent members will increment from the assigned value unless explicitly defined.

Example:

```
enum ResponseCode {
    Success = 200,
    BadRequest = 400,
    NotFound = 404
}

console.log(ResponseCode.Success);    // Output: 200
console.log(ResponseCode.BadRequest); // Output: 400
console.log(ResponseCode.NotFound);   // Output: 404
```

Key Points:

- Members can have arbitrary numeric values.
- There’s no restriction on the order of values.

---

3. String Enums

Enums can also store strings, which are often more descriptive.

Example:

```
enum Direction {
    North = "NORTH",
    South = "SOUTH",
    East = "EAST",
    West = "WEST"
}

console.log(Direction.North); // Output: "NORTH"
console.log(Direction.West);  // Output: "WEST"
```

Key Points:

- Each member must have a manually assigned string value.
- Unlike numeric enums, string enums do not auto-increment.

---

4. Heterogeneous Enums

Enums can mix numeric and string values, but this is not recommended for readability.

Example:

```
enum Mixed {
    Yes = 1,
    No = "NO"
}

console.log(Mixed.Yes); // Output: 1
console.log(Mixed.No);  // Output: "NO"
```

Key Points:

- Mixing types can make the enum harder to understand and maintain.

Reverse Mapping (Only for Numeric Enums)

Numeric enums support reverse mapping, allowing you to get the name of the enum member using its value.

Example:

```
enum Status {
    Pending = 1,
    InProgress,
    Completed
}

console.log(Status[1]); // Output: "Pending"
console.log(Status[2]); // Output: "InProgress"
```

Key Points:

- Reverse mapping works only for numeric enums, not string enums.

Enum as Types

Enums can be used to type variables or parameters to ensure they have only valid values.

Example:

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

function assignRole(role: Role) {
    if (role === Role.Admin) {
        console.log("Admin access granted.");
    }
}

assignRole(Role.Admin); // Output: Admin access grante
d.
```

```
// assignRole(1);        // Error in strict mode if Rol
e is not directly referenced
```

Use Cases of Enums

- **Readable Constants**: Replace magic numbers or strings with meaningful names.
- **Grouping Values**: Group related constants like roles, directions, or states.
- **Type Safety**: Prevent invalid values from being assigned.

Best Practices

1. **Use String Enums**: Prefer string enums for descriptive and human-readable values.
2. **Avoid Heterogeneous Enums**: They can lead to confusion.
3. **Use Reverse Mapping Only If Needed**: Numeric enums support it, but it’s rarely required.

Enums are a powerful feature that helps you organize related constants in a type-safe and maintainable way. Let me know if you'd like more detailed scenarios or examples!

▼ 8. **Type Aliases**

Type aliases in TypeScript allow you to create a custom name for a type. This makes your code more readable and reusable, especially when working with complex types.

Syntax of Type Alias

To create a type alias, use the `type` keyword followed by the alias name and the type definition.

```
type AliasName = TypeDefinition;
```

Basic Example

You can create a type alias for a single type.

```
type Username = string;

const user1: Username = "Alice";
const user2: Username = "Bob";

console.log(user1); // Output: Alice
console.log(user2); // Output: Bob
```

Explanation:

- `Username` is a type alias for the `string` type.
- Any variable of type `Username` must hold a string value.

Type Alias for Union Types

You can use type aliases to simplify union types.

```
type Status = "Pending" | "InProgress" | "Completed";

function updateStatus(status: Status): void {
    console.log(`Status updated to: ${status}`);
}

updateStatus("Pending");      // Output: Status updated
to: Pending
updateStatus("Completed");    // Output: Status updated
to: Completed
// updateStatus("Unknown");   // Error: Argument is not
assignable to type 'Status'.
```

Explanation:

- `Status` is a type alias for a union type (`"Pending" | "InProgress" | "Completed"`).
- It enforces that only these specific string values are allowed.

Type Alias for Objects

You can create a type alias for complex object structures.

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

const user: User = {
    id: 1,
    name: "Alice",
    isActive: true
};

console.log(user); // Output: { id: 1, name: "Alice", i
sActive: true }
```

Explanation:

- `User` is a type alias for an object with `id`, `name`, and `isActive` properties.
- Every object of type `User` must follow this structure.

Type Alias for Arrays

You can also use type aliases to define the structure of arrays.

```
type StringArray = string[];
type NumberArray = number[];

const names: StringArray = ["Alice", "Bob"];
const scores: NumberArray = [10, 20, 30];

console.log(names);  // Output: ["Alice", "Bob"]
console.log(scores); // Output: [10, 20, 30]
```

Explanation:

- `StringArray` is a type alias for an array of strings.
- `NumberArray` is a type alias for an array of numbers.

Type Alias vs Interface

While both `type` and `interface` are used for defining object types, there are some differences:

Feature	Type Alias	Interface
Objects	Supported	Supported
Union Types	Supported	Not Supported
Extensions	Cannot extend, but can combine	Can extend other interfaces

Example: Type Alias with Union vs Interface

```
// Type Alias for Union
type Status = "Pending" | "Completed";

// Interface (Not Possible with Union)
interface IStatus {
    status: "Pending" | "Completed";
}
```

When to Use?

- Use `type` for unions or more complex types.
- Use `interface` when extending or implementing is required.

Key Points to Remember

1. **Custom Name:** A type alias creates a custom name for any type.
2. **Simplify Code:** Use it for long or complex types to make the code cleaner.
3. **Reusable:** Aliases can be reused throughout your code.
4. **Not Extendable:** Type aliases can’t extend other types directly like interfaces.

By using type aliases effectively, you can write cleaner, more maintainable TypeScript code. Let me know if you'd like to explore specific scenarios!

▼ 9. **Classes and Access Modifiers**:

Classes are a blueprint for creating objects. They encapsulate data and behavior together, making code more organized and reusable. TypeScript enhances classes from JavaScript by introducing **access modifiers** to control the visibility of properties and methods.

Classes in TypeScript

A class in TypeScript is defined using the `class` keyword. You can include:

- **Properties**: Variables tied to the class.
- **Methods**: Functions tied to the class.

Basic Example of a Class

```
class Person {
    name: string; // Property
    age: number;

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

    greet(): void { // Method
        console.log(`Hello, my name is ${this.name} and
I am ${this.age} years old.`);
    }
}

const person1 = new Person("Alice", 25);
person1.greet(); // Output: Hello, my name is Alice and
I am 25 years old.
```

Explanation:

- `constructor`: A special method to initialize the properties of the class.
- `person1`: An object created from the `Person` class.

Access Modifiers

Access modifiers control the visibility of properties and methods. There are three types:

1. `public` (Default)

- Properties/methods are accessible from anywhere (inside or outside the class).

Example:

```
class Car {
    public brand: string;

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

    public drive(): void {
        console.log(`${this.brand} is driving.`);
    }
}

const car = new Car("Toyota");
console.log(car.brand); // Output: Toyota
car.drive(); // Output: Toyota is driving.
```

---

2. `private`

- Properties/methods are only accessible within the class.

Example:

```
class BankAccount {
    private balance: number;

    constructor(initialBalance: number) {
        this.balance = initialBalance;
    }

    deposit(amount: number): void {
        this.balance += amount;
        console.log(`Deposited: ${amount}. New balance:
${this.balance}`);
    }

    private showBalance(): void { // Not accessible out
side
        console.log(`Balance: ${this.balance}`);
    }
```

```
}

const account = new BankAccount(1000);
account.deposit(500); // Output: Deposited: 500. New ba
lance: 1500
// account.balance;   // Error: Property 'balance' is p
rivate
// account.showBalance(); // Error: Method 'showBalanc
e' is private
```

---

3. `protected`

- Similar to `private`, but accessible in the class and its subclasses.

Example:

```
class Employee {
    protected salary: number;

    constructor(salary: number) {
        this.salary = salary;
    }
}

class Manager extends Employee {
    showSalary(): void {
        console.log(`Manager's salary is: ${this.salar
y}`);
    }
}

const manager = new Manager(50000);
manager.showSalary(); // Output: Manager's salary is: 5
0000
// manager.salary;    // Error: Property 'salary' is pr
otected
```

Read-Only Properties

You can declare properties as `readonly`, meaning their value can be set only once (during initialization or in the constructor) and cannot be

changed later.

Example:

```
class Book {
    readonly title: string;

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

const book = new Book("TypeScript Guide");
console.log(book.title); // Output: TypeScript Guide
// book.title = "New Title"; // Error: Cannot assign to
'title' because it is a read-only property
```

Getters and Setters

Getters and setters allow controlled access to private properties.

Example:

```
class Student {
    private _name: string;

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

    get name(): string { // Getter
        return this._name;
    }

    set name(value: string) { // Setter
        if (value.length < 3) {
            console.log("Name is too short.");
        } else {
            this._name = value;
        }
    }
```

```
}

const student = new Student("John");
console.log(student.name); // Output: John
student.name = "Al";       // Output: Name is too shor
t.
student.name = "Alice";    // No output, value updated
console.log(student.name); // Output: Alice
```

Static Properties and Methods

Static members belong to the class itself, not to any specific object instance. They are accessed using the class name.

Example:

```
class MathUtils {
    static PI = 3.14;

    static calculateCircleArea(radius: number): number
{
        return this.PI * radius * radius;
    }
}

console.log(MathUtils.PI); // Output: 3.14
console.log(MathUtils.calculateCircleArea(5)); // Outpu
t: 78.5
```

Summary of Key Concepts

1. **Public, Private, Protected:** Control access to properties and methods.
2. **Read-Only Properties:** Ensure values cannot be modified after initialization.
3. **Getters and Setters:** Control how private properties are accessed and modified.
4. **Static Members:** Define methods or properties tied to the class, not objects.

5. **Inheritance:** Use `protected` for properties that subclasses need to access.

▼ 10. **Non-Nullable Types**

Non-Nullable Types in TypeScript

In TypeScript, **non-nullable types** help you prevent assigning `null` or `undefined` to variables or properties that require a valid value. This ensures better runtime safety and avoids common errors caused by null or undefined values.

Understanding Non-Nullable Types

By default, TypeScript allows variables to be assigned `null` or `undefined` unless strict null checks are enabled (`strictNullChecks: true` in `tsconfig.json`). When strict null checks are on:

1. A value explicitly typed as `null` or `undefined` can only hold those values.
2. Other types like `string`, `number`, or `boolean` no longer accept `null` or `undefined` as valid values.

How to Enable Strict Null Checks

To ensure non-nullable behavior, enable `strictNullChecks` in the `tsconfig.json` file:

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

Examples of Non-Nullable Types

Default Behavior Without `strictNullChecks`

```
let name: string = "Alice";
name = null; // Allowed if `strictNullChecks` is false
name = undefined; // Allowed if `strictNullChecks` is f
alse
```

With `strictNullChecks`

```
let name: string = "Alice";
// name = null; // Error: Type 'null' is not assignable
to type 'string'
// name = undefined; // Error: Type 'undefined' is not
assignable to type 'string'
```

Using `NonNullable` Utility Type

TypeScript provides a built-in `NonNullable` utility type to explicitly exclude `null` and `undefined` from a type.

Syntax

```
NonNullable<Type>
```

Example:

```
type NullableString = string | null | undefined;
type NonNullableString = NonNullable<NullableString>;

let value: NullableString;
value = "Hello"; // Allowed
value = null;    // Allowed

let strictValue: NonNullableString;
strictValue = "Hello"; // Allowed
// strictValue = null; // Error: Type 'null' is not ass
ignable to type 'string'
// strictValue = undefined; // Error: Type 'undefined'
is not assignable to type 'string'
```

Practical Scenarios for Non-Nullable Types

Avoiding Null Errors in Functions

When working with functions, non-nullable types can ensure inputs and return values are valid.

Example Without Non-Nullable Types

```
function greet(name: string | null): void {
    console.log(`Hello, ${name?.toUpperCase()}`); // Ri
sk of null or undefined
}

greet(null); // Output: Hello, undefined
```

Example With Non-Nullable Types

```
function greet(name: NonNullable<string>): void {
    console.log(`Hello, ${name.toUpperCase()}`);
}

// greet(null); // Error: Type 'null' is not assignable
to type 'string'
greet("Alice"); // Output: Hello, ALICE
```

Ensuring Non-Nullable Parameters

```
function getLength(value: string | null): number {
    if (value === null) {
        throw new Error("Value cannot be null");
    }
    return value.length;
}

getLength(null); // Throws an error
```

With the `NonNullable` type:

```
function getLength(value: NonNullable<string>): number
{
    return value.length;
}

// getLength(null); // Error: Type 'null' is not assign
```

```
able to type 'string'
getLength("Hello"); // Output: 5
```

Nullable vs. Non-Nullable in Complex Types

Non-Nullable types are particularly useful when dealing with complex objects or APIs where certain fields might be optional.

Example with Objects

```
interface User {
    name: string;
    age?: number | null; // `age` is optional and nulla
ble
}

const user: User = { name: "Bob", age: null }; // Allow
ed

function getUserAge(user: User): NonNullable<number> {
    if (user.age == null) {
        throw new Error("Age is not available");
    }
    return user.age;
}

getUserAge(user); // Throws an error if `age` is null
```

Key Points to Remember

1. `strictNullChecks` ensures that `null` and `undefined` are not assignable to other types.
2. Use the `NonNullable` utility type to explicitly exclude `null` and `undefined`.
3. Non-nullable types enhance code safety by avoiding runtime errors caused by unexpected `null` or `undefined`.
4. Apply `NonNullable` types for inputs, outputs, and object properties where values must always exist.