Strong Typing

//strongly typed syntax

```
let a : string = "Pakistan";
a = "USA";
let b : number = 9;
let c : boolean = true;
```

//type inference

```
let e = "USA"; // Typescript assume USA as String
let f = 10.9; // Typescript assume 10.9 as number
f = 22;
let g = false;  Typescript assume false as Boolean;
g = true;
```

Types can also appear in many more *places* than just type annotations. As we learn about the types themselves, we'll also learn about the places where we can refer to these types to form new constructs.

The primitives Type: string, number, and Boolean

- string represents string values like "Hello, world"
- number is for numbers like 42.
- Boolean is for the two values true and false

String Example

```
let message: string = "Hello, TypeScript!"; console.log(message); // Output: Hello, TypeScript!
```

Number Example

```
let count: number = 42; console.log(count); // Output: 42
```

Boolean Example

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

Putting It All Together

```
let greeting: string = "Welcome to TypeScript!";
let year: number = 2024;
let isLeapYear: boolean = true;
console.log(greeting); // Output: Welcome to TypeScript!
console.log(`Year: ${year}`); // Output: Year: 2024
console.log(`Is Leap Year: ${isLeapYear}`); // Output: Is Leap Year: true
```

Any

TypeScript also has a special type, any, that you can use whenever you don't want a particular value to cause typechecking errors.

```
let obj: any = {x: 0};
// None of the following lines of code will throw compiler errors.
// Using `any` disables all further type checking, and it is assumed
// you know the environment better than TypeScript.
obj.foo(); obj(); obj.bar = 100; obj = "hello"; const n: number = obj;
```

Arrays

To specify the type of an array like [1, 2, 3], you can use the syntax number[]; this syntax works for any type (e.g. string[] is an array of strings, and so on).

String Array:

```
let fruits: string[] = ["apple", "banana", "cherry"];
console.log(fruits); // Output: ["apple", "banana", "cherry"]
```

Number Array:

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

Boolean Array:

```
let isChecked: boolean[] = [true, false, true];
console.log(isChecked); // Output: [true, false, true]
```

Array of Objects

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

let users: User[] = [
   { name: "Alice", age: 30, isActive: true },
   { name: "Bob", age: 25, isActive: false }];

console.log(users);
// Output: [{ name: "Alice", age: 30, isActive: true }, { name: "Bob", age: 25, isActive: false }]
```

Multi-dimensional Arrays

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

console.log(matrix);
// Output: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Type Annotations

Type annotations in TypeScript are a way to explicitly specify the types of variables, function parameters, return values, and object properties. They help catch errors at compile time, improve code readability, and provide better tooling support.

Type Annotations on variable

```
let age: number = 25;
let name: string = "Alice";
let isActive: boolean = true;
```

Type Annotations on functions Parameters

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

greet("Alice"); // Output: Hello, Alice
```

Type Annotations on Function Return Types

```
function add(a: number, b: number): number { return a + b; }
let result = add(5, 3); // result is inferred to be of type number
```

Type Annotations on Objects

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

let user: User = {
  name: "Alice",
  age: 30,
  isActive: true};
```

Type Annotations on Arrays

```
let scores: number[] = [90, 85, 88];
let fruits: Array<string> = ["apple", "banana", "cherry"];
```

Type Annotations on Tuples

```
let tuple: [string, number, boolean] = ["Alice", 30, true];
```

Type Annotations on Type Aliases

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

Types Union

Union types in TypeScript allow you to define a variable that can hold multiple types. They are useful when a value can be of more than one type, providing flexibility while maintaining type safety.

Basic Syntax
You can create a union type by using the pipe (|) symbol between the types.

```
let value: string | number;
value = "Hello";
console.log(value);  // Output: Hello
value = 42;
console.log(value);  // Output: 42
```

In this example, value can be either a string or a number

Function Parameters

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

printId(101);  // Output: ID: 101
printId("202");  // Output: ID: 202
```

The printId function can accept an argument that is either a string or a number.

Arrays with Union Types

```
let mixedArray: (string | number)[] = ["apple", 1, "banana", 2];
console.log(mixedArray);  // Output: ["apple", 1, "banana", 2]
```

In this example, mixedArray can hold both string and number values.

Type Narrow

To work with union types effectively, you often need to narrow down the type within a certain scope. TypeScript provides several ways to narrow types:

Type Guards

Using typeof to check the type at runtime:

```
function printId(id: string | number): void {
   if (typeof id === "string") {
      console.log(`ID (string): ${id.toUpperCase()}`);
   } else {
      console.log(`ID (number): ${id}`);    }}

printId(101);  // Output: ID (number): 101
printId("abc");  // Output: ID (string): ABC
```

Type Aliases

Type aliases in TypeScript allow you to create a new name for a type. This can be particularly useful for simplifying complex type definitions, improving code readability, and making it easier to maintain and reuse types across your codebase.

Basic Syntax

You can define a type alias using the type keyword followed by the name of the alias and an equal sign = to assign it to a type.

Basic Usage

```
type StringOrNumber = string | number;

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

In this example, String Or Number is a type alias for the union type string | number.

Function Parameters

```
type ID = string | number;

function printId(id: ID): void {
   console.log(`ID: ${id}`);}

printId(101);  // Output: ID: 101
printId("202");  // Output: ID: 202
```

Objects

```
type Point = {
   x: number;
   y: number;
};

let point: Point = { x: 10, y: 20 };
console.log(point);  // Output: { x: 10, y: 20 }
```

Type Interfaces

In TypeScript, interfaces are used to define the shape of an object, specifying the structure and types of its properties. Interfaces can be used to type-check the shape of objects, ensuring they conform to a specific structure. Unlike type aliases, interfaces can be extended and implemented, offering more flexibility for object-oriented programming

Basic Syntax
You can define an interface using the interface keyword followed by the name of the interface.

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

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

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

In this example, the Person interface defines an object with name and age properties. The person variable is then typed as Person.

Optional Properties

You can use the ? symbol to denote optional properties in an interface.

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

let person1: Person = { name: "Alice" };
let person2: Person = { name: "Bob", age: 25 };

console.log(person1);  // Output: { name: "Alice" }
console.log(person2);  // Output: { name: "Bob", age: 25 }
```

In TypeScript, both type aliases and interfaces can be used to define the shape of objects and other types. However, there are differences in their capabilities and use cases. Here are the key differences between type aliases and interfaces:

Declaration

- **Type Aliases**:
  - Created using the type keyword.
    Can define primitive types, union types, tuple types, function types, and object types.
    ```
    type StringAlias = string;
    type NumberOrString = number | string;
    type Point = { x: number; y: number };
    ```

- **Interfaces**:

  Created using the interface keyword.
  Primarily used to define the shape of objects.
  ```
  interface Point {
     x: number;
     y: number;}
  ```
- Use **interfaces** when you need to define the structure of an object and expect that structure to be extended or implemented.
- Use **type aliases** for primitives, unions, intersections, tuples, and when you need more complex type combinations.

Types Literal

Type literals in TypeScript define types with specific, exact values rather than general types like string or number. They are useful when you need a variable to have only a certain set of values, offering more precise type-checking. This can be particularly handy for enums, configuration settings, and function arguments that accept limited values.

Basic Syntax

Type literals can be string literals, number literals, or boolean literals.

String Literal Types

String literal types allow you to specify that a variable can only take on one of a specified set of string values.

```
type Direction = "north" | "south" | "east" | "west";

let move: Direction;

move = "north"; // Valid
move = "east";  // Valid
// move = "up"; // Error: Type '"up"' is not assignable to type 'Direction'.
```

Number Literal Types

Similarly, number literal types restrict a variable to specific numeric values.

```
type StatusCode = 200 | 404 | 500;

let status: StatusCode;

status = 200; // Valid
status = 404; // Valid
// status = 300; // Error: Type '300' is not assignable to type 'StatusCode'.
```

Boolean Literal Types

Boolean literal types specify that a variable can only be `true` or `false`.

```
type YesOrNo = true | false;
let answer: YesOrNo;

answer = true;  // Valid
answer = false; // Valid
// answer = "true"; // Error: Type '"true"' is not assignable to type 'YesOrNo'.
```

Type intersections

Type intersections in TypeScript allow you to combine multiple types into one. The resulting type has all the properties of the combined types. This is useful for creating types that need to satisfy multiple constraints or for merging the properties of different types.

Basic Syntax

The syntax for creating an intersection type uses the `&` (ampersand) symbol.

```
type A = { name: string };

type B = { age: number };

type AB = A & B;

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

In this example, AB is an intersection type that combines A and B. The person variable must have both name and age properties.

Combining Object Types

Intersection types are often used to combine object types, ensuring that the resulting type includes properties from all the combined types.

```
type Address = {
   street: string;
   city: string;};

type Contact = {
   phone: string;
   email: string;};

type Person = Address & Contact;

let contactPerson: Person = {
   street: "123 Main St",
   city: "Anytown",
   phone: "123-456-7890",
   email: "example@example.com"};

console.log(contactPerson);
```

null and **undefined**

In TypeScript, null and undefined are used to represent the absence of a value. They are distinct types and values with their own specific meanings and use cases.

null

Type: null

Description: Represents an intentional absence of any object value.

Use Case: Commonly used to signify a missing or intentionally empty object.

```
let value: null = null;
```

undefined

Type: `undefined`

Description: Indicates that a variable has been declared but has not yet been assigned a value.

Use Case: Default value for variables that are declared but not initialized.

```
let value: undefined = undefined;
```

Differences Between null and undefined

Origin:

null is an intentional value indicating no value.
undefined is the default value for uninitialized variables.

TypeScript Types:

Both null and undefined have their own types, but they are also subtypes of void and any.

- null represents an intentional absence of value.
- undefined indicates a variable has been declared but not initialized.
- Use union types to allow null and undefined in type annotations.
- Optional properties and parameters implicitly include undefined.
- The non-null assertion operator (!) can be used to assert that a value is not null or undefined.
- Enabling strictNullChecks in TypeScript ensures safer handling of null and undefined.
- Type guards can be used to safely work with null and undefined values.