**Union Types**

Annotating a variable with union types means specifying that the variable can hold values of multiple types. Union types in TypeScript are denoted using the | symbol between the types.

For example:

let myVar: string | number;

myVar = "Hello"; // Valid assignment

console.log(myVar)

myVar = 42; // Valid assignment

console.log(myVar)

In this code, myVar is annotated with a union type of string | number, which means it can hold either string or number values. You can assign values of either type to it without causing a type error. However, if you try to assign a value of a different type (e.g., boolean), TypeScript will report a type error because it's not part of the specified union type.

**Literal Types**

Literal types are a type system feature in TypeScript that allow you to specify exactly which values are allowed for a particular variable or function parameter. Instead of using a general type like string or number, you can specify a literal type that can only take on a specific value.

For example, you can define a literal type like this:

let direction: "left" | "right" | "up" | "down";

In this case, the direction variable can only be assigned one of the four literal string values: "left", "right", "up", or "down". Any other value would result in a type error.

Literal types are especially useful in scenarios where you want to be very explicit about the allowed values for a variable, parameter, or property. They provide a level of type safety by ensuring that only specific values are accepted, which can help catch errors at compile time rather than runtime.

Here's an example of using a literal type in a function parameter:

function setColor(color: "red" | "green" | "blue") {

// ...

}

In this case, the color parameter can only be one of the three specified string literals: "red", "green", or "blue". This helps prevent potential bugs by restricting the input to a predefined set of values.

Literal types can be combined with other TypeScript features like union types, intersection types, and more to create powerful and expressive type definitions in your code.

**Nullable Types**

Nullable types in TypeScript allow a variable or parameter to have either a specific data type (like string or number) or the special value null. This feature is useful for scenarios where a value might be absent or undefined, and you want to explicitly account for that possibility in your type annotations.

In TypeScript, you can create a nullable type by appending | null to an existing data type. For example:

let username: string | null = "Glicher"; // The variable can hold a string or null

let age: number | null = null; // The variable can hold a number or null

In this example, username can either store a string or null, while age can either store a number or null.

Nullable types are particularly helpful when working with data that may not always be present or initialized. They make it clear in the type system that a variable might have the value null, which can help prevent runtime errors caused by trying to access properties or methods on a null value.

Here's an example of using nullable types with function parameters:

function greetUser(username: string | null) {

if (username === null) {

console.log("Hello, Guest!");

} else {

console.log(`Hello, ${username}!`);

}

}

greetUser("Glitcher"); // Output: Hello, Glitcher!

greetUser(null); // Output: Hello, Guest!

In this function, username is defined as a nullable string, so it can accept either a string or null. The function handles both cases, providing a default greeting for guests when username is null.

Nullable types help improve code safety and clarity by explicitly indicating the possibility of missing or undefined data. However, developers should be cautious when using nullable types to avoid unnecessary null checks and ensure proper handling of null values in their code.

**Type Alias**

A **Type Alias** in TypeScript allows you to create a custom name for a type. It's a way to define your own custom data types that can be composed of existing types like numbers, strings, objects, or other custom types. Type aliases provide improved readability, maintainability, and reusability for complex type definitions.

// Define a type alias for string

type MyString = string;

// Use the type alias for string

let myName: MyString = "Glitcher";

// Define a type alias for a union of string or number

type MyStringOrNumber = string | number;

// Use the type alias for the union type

let myValue: MyStringOrNumber = 10;

In this code:

* MyString is a type alias for the string type.
* MyStringOrNumber is a type alias for a union type that can be either a string or a number.
* We then declare variables (myName and myValue) using these type aliases to assign values of the specified types.

let's explore another example, this time using a Type Alias with objects.

type Person = {

name: string;

age: number;

email?: string;

};

const alice: Person = {

name: "Alice",

age: 30,

email: "alice@example.com",

};

const bob: Person = {

name: "Bob",

age: 25,

};

console.log(alice);

console.log(bob);

// another example

type Employee = {

name: string;

age: number;

email?: string;

};

const alice: Employee = {

name: "Alice",

age: 30,

email: "alice@example.com",

};

const bob: Employee = {

name: "Bob",

age: 25,

};

console.log(alice);

console.log(bob);

In this example, we define a type alias Employee for an object with three properties: name (a required string), age (a required number), and email (an optional string). We then create two objects, alice and bob, using this Employee type alias. alice includes all three properties, while bob omits the optional email property.

Type aliases provide a convenient way to define complex object structures with specified types and reuse them throughout your code, improving code readability and maintainability.

**Intersection Type**

An intersection type in TypeScript allows you to combine multiple types into one, creating a new type that has all the properties and functionalities of the individual types being intersected.

Here's a simple example to illustrate:

type FirstType = {

name: string;

age: number;

};

type SecondType = {

address: string;

phone: string;

};

type CombinedType = FirstType & SecondType;

const person: CombinedType = {

name: 'John Doe',

age: 30,

address: '123 Main St',

phone: '123-456-7890'

};

In this example, we have two types, FirstType and SecondType. We then create a new type CombinedType by using the & (intersection) operator, which combines the properties of both FirstType and SecondType. The CombinedType includes all properties from both types, allowing you to use them together.