STEP 04

Union Types

In TypeScript, a union type allows a variable to be one of several types. You define a union type by listing the possible types separated by the vertical bar (|). Here's the basic syntax:

```
let variableName: type1 | type2 | type3;
```

Here are some examples to illustrate how to use union types:

Example 1: Basic Union Type

```
let value: number | string;
value = 42;       // valid
value = "hello";   // valid
// value = true;   // Error: Type 'boolean' is not assignable to type 'number | string'.
```

Example 2: Union Type in Function Parameters

```
function printId(id: number | string) {
   console.log(`ID: ${id}`);}
printId(101);      // Output: ID: 101
printId("202");     // Output: ID: 202
```

Example 3: Union Type in an Interface

```
interface Response {
   status: "success" | "error";
   data: string | null;}
const successResponse: Response = {
   status: "success",
   data: "Data loaded successfully"};
const errorResponse: Response = {
   status: "error",
   data: null};
```

Example 4: Using Type Guards with Union Types

Type guards are used to handle the different types in a union. You can use `typeof`, `instanceof`, or custom type guard functions.

```
function printValue(value: number | string) {
   if (typeof value === "number") {
      console.log(`Number: ${value}`);
   } else if (typeof value === "string") {
      console.log(`String: ${value}`);   }  }
printValue(123);     // Output: Number: 123
printValue("abc");   // Output: String: abc
```

Example 5: Union Type in Arrays

You can also create arrays that hold multiple types using union types.

```
let mixedArray: (number | string)[] = [1, "two", 3, "four"];
mixedArray.push(5);      // valid
mixedArray.push("six");  // valid
// mixedArray.push(true); // Error: Type 'boolean' is not assignable to type 'number | string'.
```

Example 6:

In this example, the variable `myname` is declared with a union type of `string | null`. This means `myname` can hold either a string value or `null`. The code demonstrates assigning and logging these values.

```
let myname: string | null;
myname = null;
console.log(myname);
myname = "zia";
console.log(myname);
//myname = undefined; //Error
//myname = 12; //Error
```

Example 7:

The code demonstrates how TypeScript's type system enforces type safety, particularly with union types. The variable `myAge` can hold either a string or a number, but using methods that are specific to one type (like `toLowerCase` for strings) requires type narrowing.

```
let myAge: string | number;

myAge = 16;//narrowing

console.log(myAge);

myAge = "sixteen"; // Now narrowing to string

console.log(myAge.toLowerCase()); // Output: sixteen

console.log(myAge.toString()); // common to both types

                                    //can be called even without narrowing
```

Example 8: narrowing

Initial Assignment and Conditional Expression:

```
let newAge = Math.random() > 0.6 ? "Khan": 60;
```

// The following line would cause an error because the transpiler cannot determine if newAge is a string or number at this point.

Type Narrowing Using Equality Check:

```
if (newAge === "Khan") {
console.log(newAge.toUpperCase());
```

Type Narrowing Using `typeof`:

```
if(typeof newAge === "string"){
console.log(newAge.toUpperCase());
```

Inside this block, TypeScript knows that `newAge` is a string. // Can be called, Output: KHAN

Using Ternary Operator for Type Narrowing:

```
typeof newAge === "string"
? console.log(newAge.toUpperCase()) // Ok: string, Output: KHAN
: console.log(newAge.toFixed()); // Ok: number, Output: 60.00 (or similar)
// Using a ternary operator to narrow the type
```

```
let age: number | "died" | "unknown";

age = 90;//OK
age = "died";//OK
age = "unknown";//OK
//age = "living";//Error

------------------------------------------------------------
let zia: "zia";

zia = "zia";
//zia = "khan";//Error

------------------------------------------------------------------

let yourName = Math.random() > 0.6 ? "Hira Khan": undefined;

if (yourName) {
   yourName.toUpperCase(); // Ok: string
}

//yourName.toUpperCase();//Error: Object is possibly 'undefined'.

yourName?.toUpperCase();//OK
-----------------------------------------------------------------------

// You can also define a type alias
type RawData = boolean | number | string | null | undefined;

let data: RawData;

// You can even combine them

type Id = number | string;

type IdMaybe = Id | undefined | null;
```