step04_unions_literals

```
let myname: string | null;
let myAge: string | number;
```

Here, **myname** can hold either a string or **null**, while **myAge** can hold either a string or a number.

Narrowing the Variable Types

The code demonstrates how TypeScript narrows down the type of a variable based on assignments:

```
myname = null; // Assigned null to myname, now it's explicitly of type null
myname = "zia"; // Assigned a string, now myname can only hold strings

myAge = 16; // Assigned a number, myAge can now only hold numbers
myAge = "Dont Know"; // Assigned a string, now myAge can only hold strings
```

Usage and Limitations of Narrowed Types

It showcases that once a type is narrowed down, the variable will only allow operations/methods associated with that specific type:

```
console.log(myAge.toString()); // Common method between string and number
console.log(myAge.toLowerCase()); // Can be called on a string but not on a number
```

Ternary Operator:

The ternary operator (**condition ? exprIfTrue : exprIfFalse**) is a concise way of writing an **if...else** statement in a single line. It checks a condition and returns one of two expressions based on whether the condition is **true** or **false**.

Example:

```
let result = Math.random() > 0.5 ? "Heads" : "Tails";
// If Math.random() > 0.5 is true, result will be "Heads", else "Tails"
```

It's like making a quick decision: "If this condition is true, do this; otherwise, do that."

Conditional/Typeguard Narrowing:

In TypeScript, when you perform certain checks on a variable at runtime, TypeScript's type inference becomes smarter, allowing you to access certain properties or methods based on those checks. This is known as conditional or typeguard narrowing.

Example:

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

if (newAge === "Khan") {
   newAge.toUpperCase(); // Works, as it's guaranteed to be a string here
}

if (typeof newAge === "string") {
   newAge.toUpperCase(); // Similarly works due to the runtime check
}

typeof newAge === "string"
   ? newAge.toUpperCase() // Executes for a string type
   : newAge.toFixed(); // Executes for a number type
```

Here, **newAge** initially can be either a string **"Khan"** or a number **60** based on the ternary operation. However, by checking at runtime (**typeof newAge === "string"**), TypeScript understands that when the condition is **true**, **newAge** is a string, allowing you to safely use string methods like **toUpperCase()** without causing errors.

Literal Types and Defining Specific Values

Literal types in TypeScript allow you to specify exact values that a variable can hold. It's like saying, "Hey, this variable can only hold this specific value, nothing else!"

For instance, consider this:

```
let zia: "zia";
zia = "zia"; // This works because "zia" matches the defined literal value
// zia = "khan"; // This will give an error because it doesn't match the exact defined value
```

Here, **zia** is a variable that can only store the exact string value of **"zia"**. If you try to assign anything other than **"zia"** to **zia**, TypeScript will give you an error because it has to be that specific value. It's like having a box labeled "zia" and only allowing things related to "zia" to be inside it!

This is useful when you want to be very specific about the values your variables can hold. Think of it as defining strict rules for what's allowed in certain places in your code.

Optional Chaining

Optional chaining (**?.**) is like a safety mechanism in TypeScript. It helps avoid errors when you try to access properties or methods on a variable that could be **undefined** or **null**. Instead of directly accessing the property and potentially causing an error, you use **?.** to check if the property exists before trying to access it.

Example:

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

// Without Optional Chaining
// yourName.toUpperCase(); // Error: Object is possibly 'undefined'

// With Optional Chaining
yourName?.toUpperCase(); // Safely converts to uppercase if yourName is not undefined
```

In simple terms, if **yourName** is defined, it will execute the method (**toUpperCase** in this case). If **yourName** is **undefined**, it gracefully avoids the error.

.

Type Aliases and Combination of Types

Type aliases are a way to give a name to a combination of types. It's like creating a custom type that can be reused. Combining types using **|** (union) allows a variable to have multiple possible types.

Example:

```
type RawData = boolean | number | string | null | undefined;

let data: RawData;

// Now, 'data' can be a boolean, number, string, null, or undefined.
```

Here, **RawData** is a new name for a set of types. So, **data** can be assigned values of type **boolean**, **number**, **string**, **null**, or **undefined**. This simplifies code readability when dealing with multiple types.

Optional chaining (**?.**) helps safely access properties or methods on possibly **undefined** variables, preventing errors. Type aliases (**type**) allow creating custom names for combinations of types, making code more readable and maintaining clarity when dealing with multiple types.

Step05a_objects

An object in programming is a way to organize and store data. It's like a container that holds related information in the form of key-value pairs. Each piece of data inside an object is accessed by a unique key that identifies it within that object.

Teacher Object:

```
let teacher = {
   name: "Zeeshan",
   experience: "10"
}

console.log(teacher.name); // Output: "Zeeshan"
console.log(teacher["experience"]); // Output: "10"
```

- **let teacher = {...}** creates an object named **teacher** with two properties: **name** and **experience**.
- **teacher.name** accesses the **name** property of the **teacher** object and logs its value, which is **"Zeeshan"**.
- **teacher["experience"]** accesses the **experience** property using square brackets notation and logs its value, which is **"10"**.

Student Type Declaration and Object Assignment:

```
let student: {
   name: string,
   age: number
}

student = {
   name: "Hira",
   age: 30
}

console.log(student["name"]); // Output: "Hira"
console.log(student.age); // Output: 30
```

- **let student: {...}** declares a type for the **student** object. It specifies that a **student** object should have a **name** of type **string** and an **age** of type **number**.
- **student = {...}** assigns a new object to the **student** variable, which matches the type declaration provided earlier.
- **student["name"]** accesses the **name** property of the **student** object and logs its value, which is **"Hira"**.
- **student.age** accesses the **age** property of the **student** object and logs its value, which is **30**.

This code demonstrates the creation of objects (**teacher** and **student**) in JavaScript and how to access their properties using dot notation (**object.property**) and square brackets notation (**object["property"]**). Additionally, it showcases how to define types for object properties in TypeScript (using **:** followed by the property name and its data type).

Step05b_object_aliased

Anonymous Object

```
let teacher: { name: string, exp: number } = {
    name: "Zeeshan",
    exp: 10
};
```

This code creates an object called **teacher** that has properties **name** of type **string** and **exp** of type **number**. It represents a teacher named "Zeeshan" with 10 years of experience.

2. Aliased Object Type (using Type)

```
type Student = {
    name: string,
    age?: number
};

let student: Student = {
    name: "Hira",
    age: 30
};
```

Here, we define a type **Student** using **type** that specifies an object structure with **name** as a required **string** and **age** as an optional **number**. Then, we create a **student** object of type **Student**, representing a student named "Hira" who is 30 years old.

3. Interfaces

Interfaces in TypeScript provide a way to define the structure or shape of an object. They act as a contract specifying what properties an object should have and their corresponding types.

Imagine you're creating a blueprint for different roles in a company:

Example without Interfaces:

```
let storeManager = {
    name: "Bilal",
    subordinates: 5
};
```

In this example, **storeManager** is an object representing a manager with subordinates. Now, let's say you want to define another type of manager, perhaps an online manager, without interfaces:

```
let onlineManager = {
   name: "Sarah",
   onlineTeam: true
};
```

With this approach, you're defining different managers separately, but it's hard to know the expected properties for each manager type.

Using Interfaces:

Now, let's bring in interfaces to create a blueprint for managers:

```
interface Manager {
   name: string;
   subordinates?: number;
   onlineTeam?: boolean;
}
```

With this interface:

- **name** is a required property of type **string**.
- **subordinates** is an optional property of type **number**.
- **onlineTeam** is an optional property of type **boolean**.

Now, when you create a manager:

```
let storeManager: Manager = {
   name: "Bilal",
   subordinates: 5
};
```

Or an online manager:

```
let onlineManager: Manager = {
   name: "Sarah",
   onlineTeam: true
};
```

By using the **Manager** interface, you define a structure that both **storeManager** and **onlineManager** must adhere to. It ensures consistency in the properties each manager object should have, making it easier to understand and maintain your code. Interfaces act like a contract or a guideline for creating objects with specific shapes and properties.

Differences between the following code chunks:

1. Using type to define an object structure:

```
type Student = {
   name: string,
   age?: number
};
```

2. Using direct object declaration:

```
let student: {
   name: string,
   age: number
};
```

Type vs. Direct Object Declaration:

- **Type (Using type):** It defines a reusable type named **Student** that specifies an object structure. In this case, **Student** objects have a required **name** of type **string** and an optional **age** of type **number**.
- **Direct Object Declaration:** This directly declares a specific object structure for the **student** variable. Here, **student** objects must have a required **name** of type **string** and a required **age** of type **number**. Unlike the **type** approach, the object structure is not reusable.

3.Difference between using type and interface

Using type to define an object structure:

```
type Student = {
   name: string,
   age?: number
};
```

Using an interface:

```
interface Manager {
   name: string;
   subordinates?: number;
```

```
    onlineTeam?: boolean;
}
```

Here are the key differences:

1. **Reusability:**
   - **type:** You can create a named type (e.g., **Student**) that can be reused across your codebase. This is useful when you want to define a specific shape that can be applied to multiple instances or scenarios.
   - **interface:** Similar to **type**, interfaces are also reusable. They allow you to define a structure that multiple objects can conform to. This is particularly useful for creating consistent contracts or blueprints for different parts of your code.
2. **Compatibility and Extendability:**
   - **type:** Type aliases (created using **type**) are more flexible when it comes to combining or extending types. You can use union types, intersection types, and more to create complex types.
   - **interface:** Interfaces, on the other hand, are typically used for describing the shape of objects. They can be extended but are generally more straightforward in their syntax and usage.
3. **Declaration Merging:**
   - **type:** Type aliases don't support declaration merging. If you define the same type multiple times, it will override the previous definition.
   - **interface:** Interfaces support declaration merging. If you declare the same interface multiple times, TypeScript will merge them into a single definition. This can be advantageous when working with large and modular codebases.
4. **Use Case:**
   - **type:** Use type when you need a union, intersection, or when you want to create a specific, reusable type alias.
   - **interface:** Use interfaces when you want to define the shape of an object and when you need to leverage declaration merging or extendability.

In summary, both **type** and **interface** can be used to define object structures, and the choice between them often comes down to personal preference and the specific requirements of your code. If you need more advanced features or want to create a clear, reusable type, you might choose **type**. If you want a simpler syntax and the ability to extend and merge definitions, you might choose **interface**.

Step05c_structural_typing_object_literals

In TypeScript, interfaces with similar structures can be used interchangeably due to the structural type system. Here's the code demonstrating this:

Code Explanation:

Interfaces Declaration:

```
interface Ball {
   diameter: number;
}

interface Sphere {
   diameter: number;
}
```

Both **Ball** and **Sphere** interfaces have a **diameter** property, making their structures identical.

Assigning Values:

```
let ball: Sphere = { diameter: 10 };
let sphere: Ball = { diameter: 20 };
```

Here, we're assigning objects that adhere to **Sphere** and **Ball** interfaces, respectively. Even though the assignment might seem contradictory, it works due to their similar structures.

Interchangeability:

```
sphere = ball;
ball = sphere;
```

We can assign **sphere** to **ball** and **ball** to **sphere** without any issues. TypeScript allows this because both interfaces have the same structure with a **diameter** property.

Simplified Explanation:

Think of **Ball** and **Sphere** as blueprints for creating objects. They both describe an object having a **diameter** property. When we assign a **Sphere** object to a **Ball** variable and vice versa, TypeScript allows it because they share this common property. It's like saying, "Hey, this **Sphere** object has a **diameter**, so it's okay to treat it as a **Ball**, and vice versa, because their structures match!"

This flexibility demonstrates how TypeScript evaluates compatibility based on the structure of types rather than their explicit names, allowing us to use similar structures interchangeably.

Interfaces Declaration:

```
interface Tube {
   diameter: number;
   length: number;
}
```

The **Tube** interface is defined with two properties: **diameter** and **length**.

Creating a Tube Object:

```
let tube: Tube = { diameter: 12, length: 3 };
```

We create an object **tube** that adheres to the structure defined by the **Tube** interface. It has both **diameter** and **length** properties.

Attempting to Assign a Ball to Tube (Error):

```
// Error here because the Tube type expects both diameter and length,
// while the Ball type only provides diameter. This doesn't meet the Tube's requirements.
// TypeScript catches this mismatch and raises an error.
tube = ball; // Error
```

Assigning Tube to Ball (No Error):

```
// No error here because a Ball is expected to have a diameter,
// and the Tube object provides a diameter. The length property in Tube is ignored.
ball = tube;
```

Explanation:

In the code, we define a **Tube** interface with specific properties, **diameter** and **length**. We then create a **tube** object that meets these requirements.

Next, we attempt to assign a **Ball** object to the **tube** variable, which results in an error. This is because **Tube** expects both **diameter** and **length**, but a **Ball** object only provides **diameter**.

On the other hand, assigning a **Tube** object to a **Ball** variable doesn't raise an error. TypeScript allows this because a **Ball** type expects only a **diameter**, and the **Tube** object provides that property. The **length** property in **Tube** is simply ignored in this context.

This showcases how TypeScript ensures that assignments adhere to the expected structure of types, catching mismatches and enforcing compatibility based on the defined interfaces.

Difference b/w Fresh and Steal

Fresh:

Fresh refers to a new value assigned to a variable whenever you want, creating a new instance or updating its value.

```
let mytype = { name: 'zia', id: 1 }; // Fresh
mytype = { id: 2, name: 'khan' }; // Still Fresh
```

In the example, **mytype** initially holds an object with a name and an id. When you assign a new object to **mytype** with different values for name and id, it's still considered fresh because you're providing a completely new value to the variable.

Stale:

Stale refers to reusing or assigning an object that was declared previously or has been used before.

```
let mytype2 = { name: 'ali', id: 1 }; // Fresh
mytype = mytype2; // Stale
```

In this scenario, **mytype2** is fresh as it's a new object. However, when you assign **mytype2** to **mytype**, it becomes stale because you're reusing the object that was created earlier or used elsewhere. It's like reheating yesterday's food; it's no longer fresh.

Summary:

- **Fresh** objects are newly created or updated instances assigned to a variable.
- **Stale** objects are reused or assigned objects that were created earlier or used elsewhere, losing their freshness as they're being recycled.

Fresh Object Literal Assignment (Case By Case):

Case 1:

```
let myType = { name: "Zia", id: 1 };
myType = { id: 2, name: "Tom" }; // This is okay because it matches the same properties.
```

Here, you're updating **myType** with a new object. It works fine because the properties (**id** and **name**) in the new object match those in **myType**.

Case 2a:

```
myType = { id: 2, name_person: "Tom" }; // Error! This object includes a property not defined in
the initial declaration.
```

This results in an error because the object has a property (**name_person**) that wasn't part of the original **myType** declaration. It's either a misspelling or an extra property.

Case 2b:

```
var x: { id: number, [x: string]: any }; // An index signature allowing any string property.
x = { id: 1, fullname: "Zia" };  // This works due to the index signature permitting additional properties.
```

In Case 2b, we've defined **x** with an index signature. It says: "Hey, **x** can have any property with a string name." So, assigning an object (**{ id: 1, fullname: "Zia" }**) to **x** with an extra property (**fullname**) is okay since it matches the index signature's rule.

Case 3:

```
myType = { id: 2, name: "Tom", age: 22 }; // Error! This object has an extra property (`age`).
```

This results in an error because the object includes an extra property (**age**) that wasn't initially defined in **myType**. It violates the rule where the object should only have properties that were in the original declaration.

Summary:

- **Case 1**: Updating a variable with an object having the same properties works fine.
- **Case 2a**: Error occurs when an object includes a property not in the original declaration.
- **Case 2b**: Using an index signature allows additional properties to be added to an object.
- **Case 3**: Error arises when an object includes extra properties not in the original declaration.

Stale Object Literal Assignment (Case By Case):

Case 1:

```
let myType2 = { id: 2, name: "Tom" };
myType = myType2; // Works fine because myType2 has the same properties as myType.
```

Here, **myType2** is assigned to **myType**. Since both objects have the same properties (**id** and **name**), the assignment is successful.

Case 2a:

```
let myType3 = { id: 2, name_person: "Tom" };
myType = myType3; // Error! myType3 has a property not present in myType.
```

An error occurs because **myType3** includes a property (**name_person**) that wasn't initially part of **myType**. It violates the rule where the assigned object should have properties identical to the original declaration.

```
var x: { id: number, [x: string]: any };
var y = { id: 1, fullname: "Zia" };
x = y; // Works fine due to an index signature permitting extra properties.
```

In Case 2b, an index signature in **x** allows extra properties. **y** has a property (**fullname**) not defined in **x**, but it matches the index signature's rule, so the assignment works.

Case 3:

```
var myType4 = { id: 2, name: "Tom", age: 22 };
myType = myType4; // Works fine! Stale object can have extra properties.
```

No error occurs because **myType4** includes an extra property (**age**), but when assigned to **myType**, TypeScript allows this as it's a case of using a stale object where extra properties are permitted.

Summary:

- **Case 1**: Successful assignment as both objects have the same properties.
- **Case 2a**: Error due to a property in the assigned object not found in the original.
- **Case 2b**: Works due to an index signature permitting extra properties in the assigned object.
- **Case 3**: Successful assignment because a stale object allows extra properties compared to the original declaration.

Error - x Assignment:

```
var x: { foo: number };
x = { foo: 1, baz: 2 };  // Error, excess property `baz`
```

In this case, an error occurs because the object assigned to **x** contains an extra property (**baz**) that wasn't declared in the type definition.

Error - **y** Assignment:

```
var y: { foo: number, bar?: number };
y = { foo: 1, baz: 2 };  // Error, excess or misspelled property `baz`
```

This assignment results in an error because the object assigned to **y** not only contains an extra property (**baz**) but also could potentially have a misspelled property (**baz**) compared to the defined type structure.

No Error - **a** Assignment:

```
var a: { foo: number };
var a1 = { foo: 1, baz: 2 };
a = a1; // No Error
```

There's no error here because **a1** contains an extra property (**baz**), but TypeScript doesn't raise an error. It's allowed because the excess property isn't affecting the object assigned to **a**. TypeScript's structural typing allows this assignment since **a** only requires a **foo** property, which **a1** contains.

No Error - **z** Assignment:

```
var z: { foo: number, bar?: number };
var z1 = { foo: 1, baz: 2 };
z = z1; // No Error
```

Similar to the previous case, there's no error because although **z1** contains an extra property (**baz**), it doesn't affect the assignment to **z**. The structure of **z1** satisfies the requirements of **z** as it has at least the **foo** property expected by **z**, plus an optional **bar** property.

Summary:

- **x Assignment**: Error due to an excess property in the assigned object compared to the defined type structure.
- **y Assignment**: Error due to both an excess and potentially misspelled property in the assigned object.
- **a Assignment**: No error as the excess property in **a1** doesn't affect the required structure of **a**.
- **z Assignment**: No error as the structure of **z1** satisfies the required structure of **z** even though it contains an excess property.

Step05d_nested_objects

Nested Objects

In TypeScript, nested objects refer to objects within other objects. This code defines a **Book** type that includes an **author** property, which itself is an object of type **Author**. This way, the **Book** object contains an **author** property that's structured according to the **Author** type.

Code Explanation:

1. Creating Types for Author and Book:

```
type Author = {
   firstName: string;
   lastName: string;
};

type Book = {
   author: Author;
   name: string;
};
```

Here, **Author** and **Book** are defined as types. **Author** has properties for the first and last name, while **Book** has an **author** property of type **Author** and a **name** property.

2. Assigning Values:

```
const myBook: Book = {
   author: {
      firstName: "Zia",
      lastName: "Khan"
   },
   name: "My Best Book"
};
```

- **myBook** is declared as a **Book** type.
- Inside **myBook**, the **author** property is assigned an object with properties **firstName** and **lastName**, following the structure defined by the **Author** type.
- Additionally, **myBook** has a **name** property assigned the value "My Best Book".

Simplified Explanation:

Think of the **Book** as a book cover. Each book (**Book** type) has an **author** (another object), which itself has a **firstName** and a **lastName**. The **myBook** object is an example of this **Book** type. It contains an **author** object with a specific first and last name, along with the book's **name**.

Step05e_intersection_types

```
interface Student {
   student_id: number;
   name: string;
}

interface Teacher {
   teacher_Id: number;
   teacher_name: string;}
```

- **Student Interface**:
    - Specifies two properties: **student_id** of type number and **name** of type string.
- **Teacher Interface**:
    - Specifies two properties: **teacher_Id** of type number and **teacher_name** of type string.

Creating an Intersection Type:

```
type intersected_type = Student & Teacher;
```

intersected_type:

- It's a type that combines or intersects the properties of both **Student** and **Teacher** interfaces.
- The **&** operator in TypeScript creates an intersection type, merging the properties from both interfaces into a single type.

Object Assignment:

```
let obj1: intersected_type = {
   student_id: 3232,
   name: "rita",
   teacher_Id: 7873,
   teacher_name: "seema",
};
```

- **obj1**:
    - Declared as **intersected_type**.
    - Initialized as an object that contains properties from both **Student** and **Teacher** interfaces.
    - Includes **student_id**, **name** from the **Student** interface, and **teacher_Id**, **teacher_name** from the **Teacher** interface, fulfilling the requirements of the **intersected_type**.

Accessing Object Properties:

```
console.log(obj1.teacher_Id);
console.log(obj1.name);
```

- **console.log(obj1.teacher_Id);**
    - Accesses the **teacher_Id** property of the **obj1** object, which is part of the **Teacher** interface.
    - Outputs the value **7873**.
- **console.log(obj1.name);**

    - Accesses the **name** property of the **obj1** object, which is part of the **Student** interface.
    - Outputs the value **"rita"**.

Summary:

The code defines two interfaces, **Student** and **Teacher**, each specifying certain properties. An intersection type (**intersected_type**) is formed by merging the properties of both interfaces. An object (**obj1**) adhering to this intersection type is then created with properties from both interfaces. Finally, specific properties of the object are accessed and printed to the console.

These changes in **tsconfig.json** and **package.json** are related to configuring TypeScript and the project setup:

Changes in tsconfig.json:

"target": "ES2020"

- Indicates the version of ECMAScript that TypeScript will target when transpiling code to JavaScript.
- **"target": "ES2020"** instructs TypeScript to generate JavaScript code compatible with ECMAScript 2020 features.

"module": "NodeNext"

"moduleResolution": "NodeNext"

- These settings determine the module system TypeScript uses and how it resolves module dependencies.
- **"module": "NodeNext"** specifies that the generated JavaScript modules will be in a format suitable for Node.js using the ESNext style.
- **"moduleResolution": "NodeNext"** configures TypeScript to resolve modules in a manner compatible with Node.js ESNext.

"outDir": "./dist"

- Specifies the output directory for transpiled JavaScript files.
- **"outDir": "./dist"** directs TypeScript to output the compiled JavaScript files into the **dist** directory.

Changes in package.json:

"type": "module"

- This line indicates that the project uses ES modules (ECMAScript modules) in its JavaScript files.

- **"type": "module"** is needed in Node.js projects to use ECMAScript modules instead of CommonJS modules.

Summary:

- The changes in **tsconfig.json** target ECMAScript 2020 features, set module resolution compatible with Node.js ESNext, and specify the output directory for transpiled code.
- In **package.json**, the addition of **"type": "module"** signifies the use of ECMAScript modules in the project's JavaScript files. These changes help configure TypeScript and ensure compatibility with ES2020 features and Node.js module systems.