Step 05

Object Types

In TypeScript, object types allow you to describe the shape of objects by specifying the types of their properties. This helps ensure that objects conform to a particular structure, enhancing type safety and predictability in your code. You can define object types using interfaces, type aliases, or inline type annotations.

Example

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

console.log(teacher.name);   // output Zeeshan
console.log(teacher["experience"]);  output 10

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

// Type Declaration
let student : {
   name: string,
   age: number}

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

console.log(student["name"]); // output  Hira
console.log(student.age); // output 30

---------------------------------------------------------------
```

Using Interfaces

An interface is a way to define a contract for an object, specifying the types of its properties. Interfaces can also extend other interfaces, allowing for reusable and composable type definitions.

```
interface Person {
   name: string;
   age: number;
   address?: string; // optional property}

const person1: Person = {
   name: "Alice",
   age: 30};

const person2: Person = {
   name: "Bob",
   age: 40,
   address: "123 Main St"};

console.log(person1);
console.log(person2);
---------------------------------------------
```

Object Using Type Aliases

Type aliases provide an alternative way to define object types. They can also be used to create more complex types, such as unions and intersections.

```
type Car = {
   make: string;
   model: string;
   year: number;};

const car1: Car = {
   make: "Toyota",
   model: "Camry",
   year: 2020};

const car2: Car = {
   make: "Honda",
   model: "Accord",
   year: 2019};

console.log(car1); output {make: "Toyota", model: "Camry", yea...}
console.log(car2); output {make: "Honda", model: "Accord", yea...}
```

Nested Object Types

```
interface Address {
   street: string;
   city: string;
   zipCode: string;}

interface Person {
   name: string;
   age: number;
   address: Address; // nested object}

const person: Person = {
   name: "Charlie",
   age: 25,
   address: {
      street: "456 Elm St",
      city: "Somewhere",
      zipCode: "12345"    }};

console.log(person) output {name: "Charlie", age: 25, address: street: "456 Elm St",}
```

Readonly Properties

You can make properties read-only, which means they cannot be modified after the object is created.

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

const user: User = {
   id: 1,
   name: "David"};

user.name = "Daniel"; // OK
// user.id = 2; // Error: Cannot assign to 'id' because it is a read-only property.
```

Index Signatures

Index signatures allow you to define objects with dynamic keys, where the keys and their values follow specific types.

```
interface StringDictionary {
   [key: string]: string;}

const dictionary: StringDictionary = {
   hello: "world",
   foo: "bar"};

console.log(dictionary["hello"]); // Output: world
```

Optional Properties

Optional properties are properties that may or may not be present in the object.

```
interface Employee {
   name: string;
   age?: number; } // optional property

const employee1: Employee = {
   name: "Eve"};

const employee2: Employee = {
   name: "Frank",
   age: 35};
```

Intersection Types

Intersection types combine multiple types into one, allowing an object to have properties from multiple types.

```
interface Mailable {    sendEmail: (email: string) => void;}

interface Logger {    log: (message: string) => void;}

type User = Mailable & Logger;

const user: User = {
   sendEmail: (email) => console.log(`Sending email to ${email}`),
   log: (message) => console.log(`Logging message: ${message}`)};

user.sendEmail("example@example.com");
user.log("User logged in");
```

step 05 b object_aliased

```
// anonymous

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

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

// Aliased Object Type
type Student = {
    name: string,
    age?: number
}

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

console.log(student["name"]);
console.log(student.age);
-----------------------------------------------

// Interfaces

interface Manager {
    name: string,
    subordiates?: number}

let storeManager: Manager = {
    name: "Bilal"}

-----------------------------------------------
```

step05c_structural_typing_object_

In TypeScript, structural typing is a way of relating types based solely on their members. Unlike nominal typing, which is based on explicit declarations and type names, structural typing considers types compatible if their structures (i.e., the properties and methods they define) are compatible. This allows for a flexible and intuitive approach to type compatibility and assignments.

Key Concepts of Structural Typing

1. **Type Compatibility:** Types are considered compatible if they have the same shape, regardless of their names or explicit declarations.
2. **Duck Typing:** If an object or type "looks like a duck and quacks like a duck," it's considered a duck. In other words, if an object has the required properties and methods, it is considered to be of a certain type.

```
// These two interfaces are completely
// transferrable in a structural type system:

interface Ball {  diameter: number;}
interface Sphere {    diameter: number;}

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

  sphere = ball;
  ball = sphere;

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

  //tube = ball;//Error
  ball = tube;

  // Because a ball does not have a length, then it cannot be assigned to the tube variable. However, all of the members of Ball are inside tube, and so it can be assigned. TypeScript is comparing each member in the type against each other to verify their equality.
```

Freshness
//Case when "FRESH"" object literal are assigned to a variable
//Rule: When a fresh object literal is assigned to a variable or passed for a parameter of a non-empty target type,
//it is an error for the object literal to specify properties that don't exist in the target type.
//The rationale for the below two errors is that since the fresh types of the object literals are
//never captured in variables, static knowledge of the excess or misspelled properties should not be silently lost.

```
let myType = { name: "Zia", id: 1 }; //Case 1: can only assign a type which has the the same properties
                                     Object literals can only have properties that exist in contextual type

myType = { id: 2,  name: "Tom" };  //Case 2a: Error, renamed or missing property

myType = { id: 2,  name: "Tom", age: 22 };//Case 3: Error, excess property
```

when STALE object literal are assigned to a variable

```
let myType2 = { id: 2,  name: "Tom" }; /Case 1: can only assign a type which has the the same properties, rule same for fresh and stale object

let myType3 = { id: 2,  name_person: "Tom" }; Case 2: Error, renamed or missing property, rule same for stale and fresh object
```

//A type can include an index signature to explicitly indicate that excess properties are permitted in with fresh objects:

```
var x: { id: number, [x: string]: any };//Note now 'x' can have any name, just that the property should be of type string
var y = { id: 1, fullname: "Zia" };
x = y;// Ok, `fullname` matched by index signature
```

Structural typing VS Duck typing

Structural typing

Definition: Structural typing in TypeScript allows types to be compatible based on their structure. Two types are considered the same if they have the same properties with matching types, regardless of their names or declarations.

Duck Typing

Definition: Duck typing is a form of structural typing where the type compatibility is determined by the presence of certain properties or methods, rather than the actual type. If an object behaves like a certain type (has the required properties), it can be treated as that type.

Summary

Concept	Structural Typing	Duck Typing
Definition	Type compatibility based on structure	Type compatibility based on behavior
Focus	Properties and types of properties	Presence of expected methods/properties
Example	Two types are interchangeable if their shape matches	Any object can be used if it implements the required properties

step05d_nested_objects

Nested objects in TypeScript refer to objects that contain other objects as their properties. This allows you to create complex data structures and represent hierarchical relationships. TypeScript's type system can enforce the shape of these nested objects, providing strong type safety.

Using Interfaces

```
interface Address {
   street: string;
   city: string;
   zipCode: string;}

interface Person {
   name: string;
   age: number;
   address: Address; // Nested object}

const person: Person = {
   name: "Alice",
   age: 30,
   address: {
      street: "123 Main St",
      city: "Somewhere",
      zipCode: "12345"    }};
```

Using Type Aliases

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

type Person = {
   name: string;
   age: number;
   address: Address; // Nested object};

const person: Person = {
   name: "Bob",
   age: 25,
   address: {
      street: "456 Elm St",
      city: "Anywhere",
      zipCode: "67890"    }};
```

step05e_intersection_types

Object Intersection Types

Object intersection types in TypeScript allow you to combine multiple object types into a single type that has all the properties of the intersected objects. This is useful for creating more complex data structures that need to adhere to multiple interfaces or shapes simultaneously.

Definition

An object intersection type is defined using the & operator. When combining types, the resulting type must have all the properties of the individual types.

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

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

type intersected_type = Student & Teacher;

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

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

step05f_any__unknown_never_type

The never type represents the type of values that never occur. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns. Variables also acquire the type never when narrowed by any type guards that can never be true.

The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, never (except never itself). Even any isn't assignable to never.

```
// Any
```

When a value is of type any, you can access any properties of it (which will in turn be of type any), call it like a function, assign it to (or from) a value of any type, or pretty much anything else that's syntactically legal:

```
let myval: any;
myval = true; // OK
myval = 42; // OK
myval = "hey!"; // OK
myval = []; // OK
myval = {}; // OK
myval = Math.random; // OK
myval = null; // OK
myval = undefined; // OK
myval = () => { console.log('Hey again!'); }; // OK
```

```
//Unknown
```

The unknown type in TypeScript represents any value but is safer than any. It is used when you want to allow any type while still enforcing type checking. With unknown, you must perform some type checking before performing operations on the value.

```
let value: unknown;
value = true; // OK
value = 42; // OK
value = "hey!"; // OK
value = []; // OK
value = {}; // OK
value = Math.random; // OK
value = null; // OK
value = undefined; // OK
value = () => { console.log('Hey again!'); }; // OK
```

```
// Assigning a value of type unknown to variables of other types

let val: unknown;
const val1: unknown = val; // OK
const val2: any = val; // OK
const val3: boolean = val; // Will throw error
const val4: number = val; // Will throw error
const val5: string = val; // Will throw error
const val6: Record<string, any> = val; // Will throw error
const val7: any[] = val; // Will throw error
const val8: (...args: any[]) => void = val; // Will throw error

// Never
```

The never type in TypeScript represents values that never occur. It is used in situations where a function or expression will never successfully complete, such as when it throws an error or has an infinite loop.

```
// Function returning never must not have a reachable end point

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

 // Inferred return type is never
function fail() {  return error("Something failed");}

 // Function returning never must not have a reachable end point

function infiniteLoop(): never {  while (true) {}}
```