Typescript

Saturday, December 14, 2024 9:44 AM

What is Object?
In TypeScript, an **object** is a data structure that allows you to store collections of related data or functionality in the form of key-value pairs.

Key Feacture :

- **Dynamic Structure**: Objects can have any number of key-value pairs.
- **Type Safety**: You can define specific types for the keys and values in an object.

```
// You can also use same as javascript(Typescript is superset of javascript).
// 1 way
let person : Object = {
    name:"Rohit",
    age:22,
    isStudent : false,
    Company : "Microsoft",
    address : {
        city:"Pune",
        country : "India"
    }
}
console.log(person);
// How to access value
console.log(person["name"]);
console.log(person["address"]["city"]);
console.log(person.address.city);
// Updation in object
person.address.city = "Mumbai";
console.log(person);
// 2way
let person1 :{name:string,age:number}={
    name : "Ram",
    age : 22
}
console.log(person1);
```

What is Type Aliases?
A **type alias** in TypeScript lets you create a shortcut or custom name for a type. It helps you make your code easier to understand and reuse by giving meaningful names to complex or repeated types.

Example :

```
type Person = {
    name:string;
    age : number;
    isStudent : boolean;
    Company : string;
    address : {
        city : string;
        country : string;
    }
};
let p1 : Person ={
    name : "Ram",
    age : 29,
    isStudent : false,
    Company:"Amazon",
    address :{
        city : "Hyderabad",
        country:"India"
```

```
    }
}
let p2 : Person ={
    name : "Krish",
    age : 35,
    isStudent : true,
    Company:"NA",
    address :{
        city : "Hyderabad",
        country:"India"
    },
}
console.log(p1);
console.log(p2);
function F1(person1:Person)
{
    return person1.name;
}
console.log(F1(p2));
```

What is Call Signature?
In TypeScript, a **call signature** is a way to define the shape of a function, specifying the arguments it accepts and the value it returns.

Use Cases

- Defining APIs for functions in a library.
- Ensuring type safety when assigning functions to variables or object properties.
- Specifying the shape of callbacks.

Example :

```
type Addfn = {
   (x:number,y:number):number; //Call Signature
};
const add : Addfn = (x:number,y:number) => x+y;
console.log(add(1,2));
```

Callback Example :

```
type Callback = (message:string)=>void;
function sample(name:string,callback:Callback)
{
   const message = `Hello ${name}`;
   callback(message);
}
function printmsg(msg:string)
{
   console.log(msg);
}
sample("Alice",printmsg);
```

What is Enum ?
In TypeScript, an **enum** (short for enumeration) is a way to define a set of named constants, either numeric or string-based. It provides a way to give more descriptive names to values, making code easier to read and maintain.

```
// Numerice ENum
enum Direction {
   up,
   down,
   left,right
}
// Custom Enum
enum Status {
   active = 1,
   inactive = 0
}
enum Role {
```

```
    Admin,
    User,
    Guest
}
// Enum with function
function getPermissions(role: Role) {
    switch (role) {
        case Role.Admin:
            return "Full access";
        case Role.User:
            return "Limited access";
        case Role.Guest:
            return "Read-only access";
    }
}
console.log(getPermissions(Role.User));
console.log(Direction);
console.log(Status.active);
```

What is Tuple?
In TypeScript, a **tuple** is a fixed-length, ordered collection of elements where each element can have a different type. Tuples are essentially arrays with predefined types and lengths.

```
type PersonInfo = [string,number,boolean];
const Person1 : PersonInfo = ["Rohit",22,true];
console.log(Person1[0]);
const displayinfo = (person:PersonInfo) :void=>{
    const [name,age,hasDriverLic] = person;
    console.log(name,age,hasDriverLic);
}
displayinfo(Person1)
// Readonly Tuple
type p1 = readonly [number];
const p2 : p1 = [1];
// p2[0] = 10; error : readonly tuple
console.log(p2[0]);
Person1[0] = "Ram"
console.log(Person1);
```

Feature	Tuple	Array
Length	Fixed length	Dynamic length
Element Types	Can have different types for each element	All elements are typically of the same type
Use Case	Used for fixed, structured data (e.g., a pair or record)	Used for collections of similar data

In TypeScript, **union types** and **intersection types** are used to combine multiple types in different ways.
1. Union (|):
A union type allows a variable to be one of several types. It means the variable can take on any one of the specified types.

```
let s1 : number | string;
s1 = 1;
s1 = "String";
console.log(s1);
type a1 = {
    name : string;
}
type a2 = {
    age : number;
}
let a3 : a1 | a2;
a3 = {name:"Ram"};
a3 = {age:22};
a3 = {name:"Rockey",age:22};//error
console.log(a3);
```

2. Intersection (&):
An intersection type combines multiple types into one. A variable with an intersection type will require all the properties of the types being combined.

```
type Person1 ={
    name: string;
    age: number;
}
type Employee ={
    employeeId: number;
}
let employee: Person1 & Employee = {
    name: "Alice",
    age: 30,
    employeeId: 1234
};  // Must have both Person and Employee properties

console.log(employee);
```

Key Differences:

- **Union (|)**: The variable can hold one of the types specified, but not both.
- **Intersection (&)**: The variable must hold all the types combined, meaning it must have all the properties of each type.

What is generics?
In TypeScript, **Generics** allow you to write flexible, reusable code by enabling functions, classes, and interfaces to work with different types while keeping the types safe. Instead of working with a single type, you can create a blueprint for various types using generics.

```
function greet<T>(arg:T):T{
    return arg;
}
console.log(greet<number>(1));
console.log(greet<string>("Hello"));
```

Function Overloading :
Same function name but different parameter.

```
function add<T, U>(a: T, b: U): void {
    console.log(a + b);
}
function add<T>(a:T):void{
    console.log(a);
}
add<number,number>(1, 2);
add<number>(1);
```

What is interface?

In TypeScript, an **interface** is a way to define the structure of an object, including the types of its properties and methods.

- **Interfaces** define the structure of objects, including properties and methods.
- They help with **type-checking** and ensure that objects adhere to a specific shape.
- **Optional** and **readonly** properties can be used to enforce more flexible or immutable data structures.
- **Interfaces can extend** other interfaces and be used in function signatures or with classes to enforce structure.

Example :

```
interface Person2{
    readonly name :string,
    age : number
}
const p3 : Person2 = {
    name : "Ram",
    age:12
}
interface Person3 {
    name: string;
    age: number;
}
```

```
interface Employee1 extends Person3 {
    employeeId: number;
}
const employee1: Employee1 = {
    name: "Alice",
    age: 30,
    employeeId: 12345
};
console.log(employee1);
console.log(p3);
```

Compiler Setting :

Target,excluse,include,root,outdir