upGrad

# Introduction to TypeScript

**Course:** Introduction to TypeScript

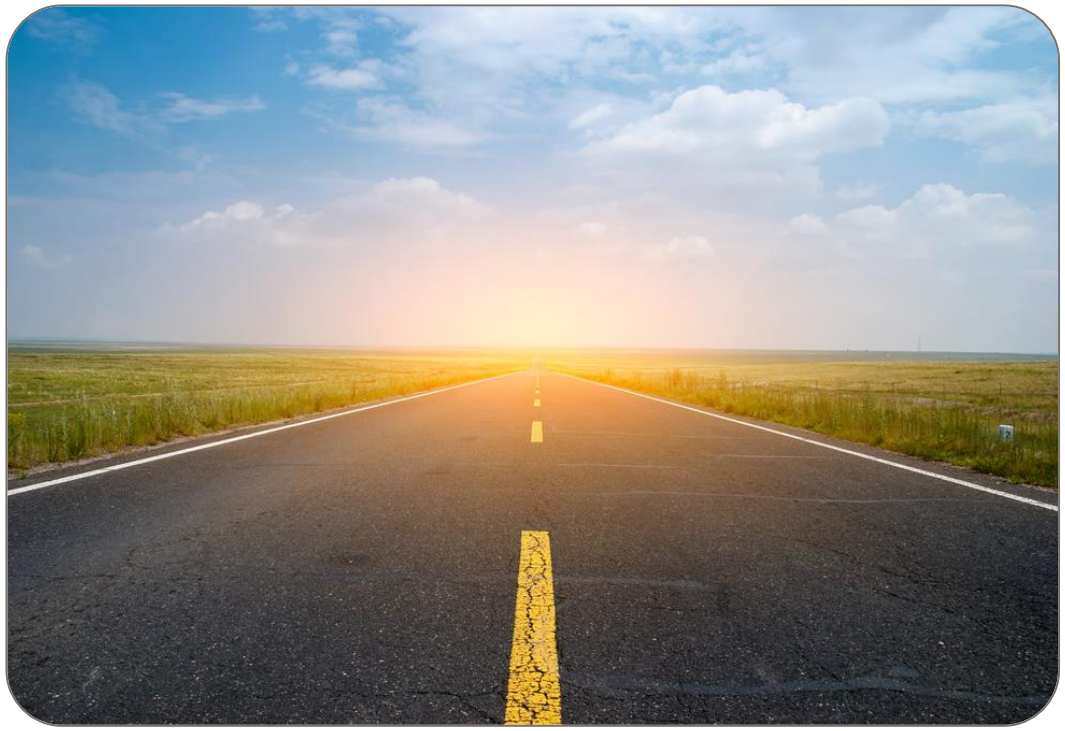
**Lecture on:** Type manipulation in TypeScript

**Instructor:** Aquib Ajani

upGrad

# COURSE ROADMAP

- Introduction to TypeScript (TS)
- Operators, Conditions and loops in TS
- Functions in TS
- Hands-on coding session - I
- Classes and interfaces in TS
- **Type manipulation in typescript**
- Modules in TS
- Hands-on coding session - II



# TODAY'S AGENDA

- Type annotation
- Type inference
- Type assertion
- Utility types
- Type manipulation

# Type Manipulation in TypeScript

# TYPE MANIPULATION IN TYPESCRIPT

## Type Annotation

- Since Typescript is a typed language, it expects a type for each variable being declared
- Various types of type annotations are being used to specify type of variables

**Basic Annotations:**

```
var height: number = 123;
var name: string = “Parth”;
var isActive: boolean = true;
```

**Type annotations in parameters:**

```
function hello(name:string){
        console.log(“hello ”+name)
}
```

# TYPE MANIPULATION IN TYPESCRIPT

## Type Annotation

**Inline Type Annotation:**

- This type of annotation allows us to declare an object for each property of the object

  **Type annotation in objects:**

```
var customer : {
id : number;
name : string;
}

Customer = {
id: 4,
name: “Parth”
};
```

# TYPE MANIPULATION IN TYPESCRIPT

## Type Inference

TypeScript is a typed language but since it is not mandatory to specify the type of variable, type inference comes into the picture, where typescript identifies the type of variable automatically

**Types are inferred by the compiler at some particular time, and those are as follows:**

- When variables are initialised
- When default values are set for parameters
- When the function return types are determined

Example:

```
var a = “hello” // type inferred is string (a:string)
var num = 2 // type inferred is number (a:number)
var b = [1,2,3,null,4] //most common type inferred which can be used for
all values like its number or null here
var c = [1,”hello”,3] // here two types of same level are there like
number and string thus union type is considered which is (string | number)
```

# Poll 1 (15 Sec)

**Will this code (1 and 2) work or throw an error?**

```
var a = [1,”hello”,3];
1–> a.push(23); 2–> a.push(true);
```

# Poll 1 (Answer)

**Will this code (1 and 2) work or throw an error?**

```
var a = [1,”hello”,3];
a.push(23); a.push(true);
```

Yes, the first one will work, as the union type is string and number

No, the second one will not work, as the union does not have Boolean as the inferred data type

# TYPE MANIPULATION IN TYPESCRIPT

## Type Assertion

Type assertion is similar to typecasting in other programming languages. However, it performs no special checking or restructuring of data, and there is no runtime effect in TS, as it is done purely by the compiler. In other programming languages such as Java, typecasting affects the performance of the program, as the knowledge of the language, compiler and the runtime environment can enhance the quality and speed in the case of such programs

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
console.log(strLength);
```

Type assertion can be performed in the following two ways:

- Using angular brackets <>
- Using the ‘as’ keyword

# TYPE MANIPULATION IN TYPESCRIPT

## Type Assertion: Example

**Basic type assertion:**

```
let vehicleNumber: string | number;
vehicleNumber = "MH-";
let code: any = 123;
let customerCode = <number> code;
console.log(typeof(customerCode)); //Output: number
```

**Type assertion with objects:**

```
let t employee = { };
employee.name = "John"; //Compiler Error: Property
'name' does not exist on type '{}'
employee.code = 123; //Compiler Error: Property
'code' does not exist on type '{}'
```

# TYPE MANIPULATION IN TYPESCRIPT

## Utility Types

Typescript has many types that help in some common type manipulations. These are listed here:

**Partial:**

```
This changes the entries to be optional in an object.
interface Data {
  name: string;
  id: number;
}
let personData: Partial<Data> = {}; // `Partial` allows name and
id to be optional
pointPart.id = 10;
```

# TYPE MANIPULATION IN TYPESCRIPT

## Utility Types

**Required:**

```
This changes the entries to be required in an object even if it is
marked as optional.
interface Bike {
  make: string;
  model: string;
  mileage?: number;
}

let myCar: Required<Car> = {
  make: 'Ford',
  model: 'Focus',
  mileage: 12000 // `Required` forces mileage to be defined
};
```

# TYPE MANIPULATION IN TYPESCRIPT

## Utility Types

**Record:**

```
This defines the type of key and value both.
Const Data : Record<string ,number>{
  ‘Parth’: 1;
}
```

# TYPE MANIPULATION IN TYPESCRIPT

## Utility Types

**Omit:**

```
Removes key from object.

interface Data {
  name: string;
  age: number;
  Place?: string;
}

const bob: Omit<Person, 'age' | 'place'> = {
  name: 'Parth'
  // `Omit` has removed age and place from the type and they can't
be defined here
};
```

# TYPE MANIPULATION IN TYPESCRIPT

## Utility Types

**Pick:**

```
Removes all the keys except the one picked.

interface Data {
  name: string;
  age: number;
  Place?: string;
}

const bob: Pick<Person, 'name' > = {
  name: 'Parth'
  // `Pick` has removed age and place from the type and they can't
be defined here
};
```

# TYPE MANIPULATION IN TYPESCRIPT

## Utility Types

**Exclude:**

```
Removes selected type from a union type.

type Data = string | number | boolean
const value: Exclude<Primitive, string> = true; // a string
cannot be used here since Exclude removed it from the type.
```

**ReturnType:**

```
gives back return type of the function.

type DataGenerator = () => { name: string; age: number; };
const point: ReturnType<PointGenerator> = {
  Name: “Parth”,
  Age: 20
};
```

# TYPE MANIPULATION IN TYPESCRIPT

## Utility Types

**Parameter:**

```
Gives the parameter types of a function type as an array.

type DataPrinter = (p: { x: number; y: number; }) => void;
const point: Parameters<DataPrinter>[0] = {
  x: 10,
  y: 20
};
```

**ReadOnly:**

```
Once the values are assigned then cannot be reassigned.

type Data = {a:string, b:string};
type ReadonlyType = ReadOnly<Data>;
```

# TYPE MANIPULATION IN TYPESCRIPT

## Utility Types

**NonNullable:**

```
Excludes null and undefined .

type Data = string | null | undefined;
type nonNullableType = NonNullable<Data>;
```

**InstanceType:**

```
Gives a type consisting of the instance type of a
constructor function.

Class Data = { x: number; y: number; }
Type Data0: InstanceType<typeof Data>; // Data0 will follow
types of Data
```

# Will this code work or throw an error?

```
function formatAmount(money: number | string) {
        let formattedAmount = "Rs. " +parseInt(money);
        console.log(formattedAmount);
        return formattedAmount;
}
```

## Will this code work or throw an error?

```
function formatAmount(money: number | string) {
        let formattedAmount = "Rs. " +parseInt(money);
        console.log(formattedAmount);
        return formattedAmount;
}
```

It will throw an error - a type mismatch will occur in the number and the string

# TYPE MANIPULATION IN TYPESCRIPT

## Type Manipulation

TypeScript, being a typed language, offers a lot of functionality, including creating types in terms of other types

The simplest example will be Generics, and various type operators can be used or the same. Some are listed here:

**Generics:**

```
function getData(arg: number): number {
  return arg;
}

here arg:number defined that the parameter will be number and the :number signifies that
return will be of number type.
Generics will make it to be Type.

function getData<Type>(arg: Type): Type {
  return arg;
}
Now when we call the function getData as
let output = getData<string>(“Parth”); // output will be string type
```

# TYPE MANIPULATION IN TYPESCRIPT

## Type Manipulation

**keyOf type operator:**

```
Its an equivalent of object.keys in JavaScript

Class Data = { x: number; y: number; }
Type keys: keyOf Data // “x”, ”y”
```

**typeOf type operator:**

```
As the name suggests, returns the type of the variable.

Class Data = “hello”;
Type keys: typeOf Data // string

typeOf is much more useful when has complex functions or
when combined with other type operators.
```

# TYPE MANIPULATION IN TYPESCRIPT

## Type Manipulation

**Indexed access types:**

```
Get specific property on another type.

type Data = { age: number; name: string; alive: boolean };
type Age = Data["age"];
```

**Conditional types:**

```
More like a ternary operator in javascript

type IamString<T> = T extends string ? 'I am string': 'I am
not string';
type str = IamString<string>; // "I am string"
type notStr = IamString<number>; // "I am not string”
```

# TYPE MANIPULATION IN TYPESCRIPT

## Type Manipulation

**Mapped types:**

```
These are widely useful when we need to derive a type another type.
// Configuration values for the current user
type AppConfig = {
  username: string;
  layout: string;
};

// Whether or not the user has permission to change configuration values
type AppPermissions = {
  [Property in keyof AppConfig as `change${Capitalize<Property>}`]: boolean
};

Here we have created a relationship using mapped type for the appconfig.
The most widely used implementation of the Mapped Types Is the
Array.prototype.map() function.
```

# TYPE MANIPULATION IN TYPESCRIPT

## Type Manipulation

**Template literal types:**

```
Expanding particular strings into combination of many strings.
The most basic example is:

type World = "world";
type Greeting = `hello ${World}`;

A little complex example will be:

type EmailIDs = "welcome_email" | "email_heading";
type FooterIDs = "footer_title" | "footer_sendoff";
type AllIDs = `${EmailIDs | FooterIDs}_id`;
type AllLocaleIDs = "welcome_email_id" | "email_heading_id" |
"footer_title_id" | "footer_sendoff_id"
```

# What will be the output of this code?

```
function example(foo: any) {
    if (typeof foo === "number") {
        // foo is type number in this block
        console.log(foo + 100);
    }

    if (typeof foo === "string") {
        // foo is type string in this block
        console.log("not a number: " + foo);
    }
}

example(2);
example(“hello”);
```

# Poll 1 (Answer)

**The output will be:**

```
123
not a number: foo
```

# KEY TAKEAWAYS

- Type Annotation
- Type Inference
- Type Assertion
- Utility Types
- Type Manipulation

# TASKS TO COMPLETE AFTER THE SESSION

| |
|---|
| MCQs |
| Coding Questions |

upGrad

# Thank you!