5

Type manipulation

Union and Intersection types, utility types

Union

- In Typescript a value can support multiple types
- We refer to each of these types as the union types
- We use union operator (**|**) syntax

```
type1 | type2 | type3 | .. | typeN
```

```
var a: number | string
```

Union

Example

```
function printId(id: number | string) {
   console.log("Your ID is: " + id);
 }

 // OK
 printId(101);
 // OK
 printId("202");

 // Error
 printId({ myID: 22342 });

//Argument of type '{ myID: number; }' is not assignable to parameter of type 'string | number'.
```

Union

- It’s common to use th same type more than once
- Use type aliases to create new types and name them

```
type ID = number | string;

var id: ID = 1
var id2: ID = "my_id"


function printId(id: ID) {
   console.log("Your ID is: " + id);
 }
```

Intersection

- An intersection type create a new type by combining multiple existing ones
- The new type has all properties of the existing types
- To combine types, use **(&)** operator

```
type typeABC = typeA & typeB & typeC & …;
```

Intersection

Example

```
interface Identity {
    id: number;
    name: string;
}
```

```
interface Contact {
   email: string;
   phone: string;
}
```

```
type Employee = Identity & Contact;
```

```
let e: Employee = {
   id: 100, //Identity
   name: 'John Doe', //Identity
   email: 'john.doe@example.com',  //Contact
   phone: '(408)-897-5684' //Contact
};
```

Template literal types

- Template literal types allows to create custom string types based on a template
- Use template interpolation syntax

```
type Greeting = `hello ${string}`;

var text: Greeting = "hello world" //OK
text = "hi world" //Error
```

- We can also use union with string literal, example:

```
type Lang = "en" | "fr" | "ar";
```

Operator: typeof

- **typeof** operator returns a string indicating the operands value type
- Can be used in expression context

```
var str = "Hello world"
console.log(typeof str) //Prints 'string'
```

- Or type context

```
var str = "Hello world"
type Custom = typeof str //Custom = string
```

Operator: keyof

- **keyof** operator is used to extract key types from an object type

```
type User = { id: number; email: number };
type P = keyof User;
//P = "id" | "email"

var p:P = "name" //Error, 'name' is not a key of User
```

Operator: instanceof

- **`instanceof`** operator checks if an object is an instance of a class
- Takes inheritance into account
- Returns true if the objects inherits from the class prorotype

```
class Person {
  name: string = '';
}

let person = new Person();
let contact = {  name: "john"}

console.log(person instanceof Person ); // true
console.log(contact instanceof Person ); // false
```

Utility types

- Typescript provides utility types that are available globaly to facilitate type transformations
- Utility types are generic types that applies to any type you provide and create new types
- Most common utility types
  - `Partial<T>`
  - `Required<T>`
  - `Pick<T>`
  - `Omit<T>`
  - `...`

Utility types

Partial<T>

- Constructs a type with all properties of T set to **optional**

```
interface Person {
    firstName: string;
    lastName: string;
}

type PersonOpt = Partial<Person>

/*
    firstName?: string;
    lastName?: string;
  */
```

Utility types

Required<T>

- Constructs a type with all properties of T set to **required**

```
interface Person {
    firstName?: string;
    lastName?: string;
}

type PersonReq = Required<Person>

/*
    firstName: string;
    lastName: string;
  */
```

Utility types

Readonly<T>

- Constructs a type with all properties of T set to **readonly**

```
var a : Readonly<Person> = {
  firstName: "Me",
  lastName: "Me"
}

a.firstName = "something" //Error: readonly
```

Utility types

Pick<T, Keys>

- Constructs a type by picking specific Keys from a type T

```
type FirstName = Pick<Person, "firstName">

/*
   firstName: string
 */
```

Utility types

Omit<T, Keys>

- Constructs a type by picking properties of type T, removing (omitting) Keys
- Opposite of Pick

```
type FirstName = Omit<Person, "firstName" >

/*
   lastName: string
 */
```

Utility types

Parameters<F>

- Return a tuple type from the types used in the parameters of a function type F

```
function f1(a: number, b: string ): void {
}

type Params = Parameters<typeof f1>
//Params : [number, string]

var params : Params = [1,"str"]
```

Utility types

ReturnType<F>

- Constructs a type from the return type of function type F

```
function f1(a: number, b: string ): void {
}


type Return = ReturnType<typeof f1>
//void
```

Utility types

UpperCase<S>

- Construct an uppercase equivalent from the string type S

```
type LANG = 'fr'| 'en' | 'ar'
type LANG_ID = Uppercase<LANG>

var lang: LANG_ID = "FR" //OK
lang = "fr" //Error
```

Utility types

LowerCase<S>

- Construct an lowercase equivalent from the string type S

```
type DIR = "RTL" | "LTR"
type dir = Lowercase<DIR>
//dir: "rtl" | "ltr"
```

Capitalize<S>

- Construct a capitalized equivalent from the string type S

```
type Dir = Capitalize<dir>
//dir: "Rtl" | "Ltr"
```