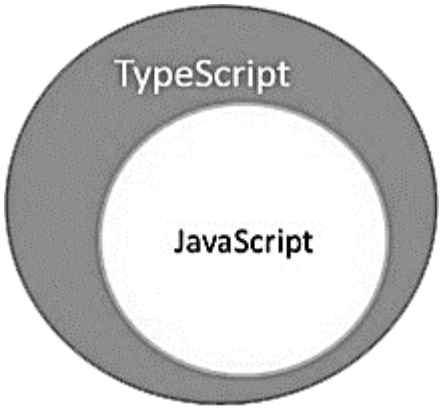
Introduction to TypeScript

Agenda

- Type Script
- Data Type
- Special Type
- Defining Type
- Utility Type
- Type Operator
- Type Guard
- Module

What is TypeScript

- TypeScript is a syntactic superset of JavaScript which adds static typing



TypeScript

- TypeScript vs JavaScript

JavaScript	TypeScript
It doesn't support strongly typed or static typing.	It supports strongly typed or static typing feature.
Netscape developed it in 1995.	Anders Hejlsberg developed it in 2012.
JavaScript source file is in ".js" extension.	TypeScript source file is in ".ts" extension.
It is directly run on the browser.	It is not directly run on the browser.
It is just a scripting language.	It supports object-oriented programming concept like classes, interfaces, inheritance, generics, etc.
It doesn't support optional parameters.	It supports optional parameters.
It is interpreted language that's why it highlighted the errors at runtime.	It compiles the code and highlighted errors during the development time.
JavaScript doesn't support modules.	TypeScript gives support for modules.
In this, number, string are the objects.	In this, number, string are the interface.
JavaScript doesn't support generics.	TypeScript supports generics.

TypeScript

- TypeScript uses compile time type checking

Static Type Checking

- TypeScript checks a program for errors before execution and does based on the kinds of values

```
const message = “hello world”;

message();              //ERROR
//This expression is not callable. Type ‘String’ has no call signature
```

```
const obj = { width: 10, height: 15 };
const area = obj.width * obj.heigth;      //ERROR

//Property 'heigth' does not exist on type '{ width: number; height:
number; }'. Did you mean 'height'?
```

Variable

- Variable can be declared using
  - var scope rules remain the same as java script
  - let scope in their containing block
    - cannot be read or write to before they are declared
    - cannot be re-declared
  - const a constant where it's value cannot be changed

Variable

- var

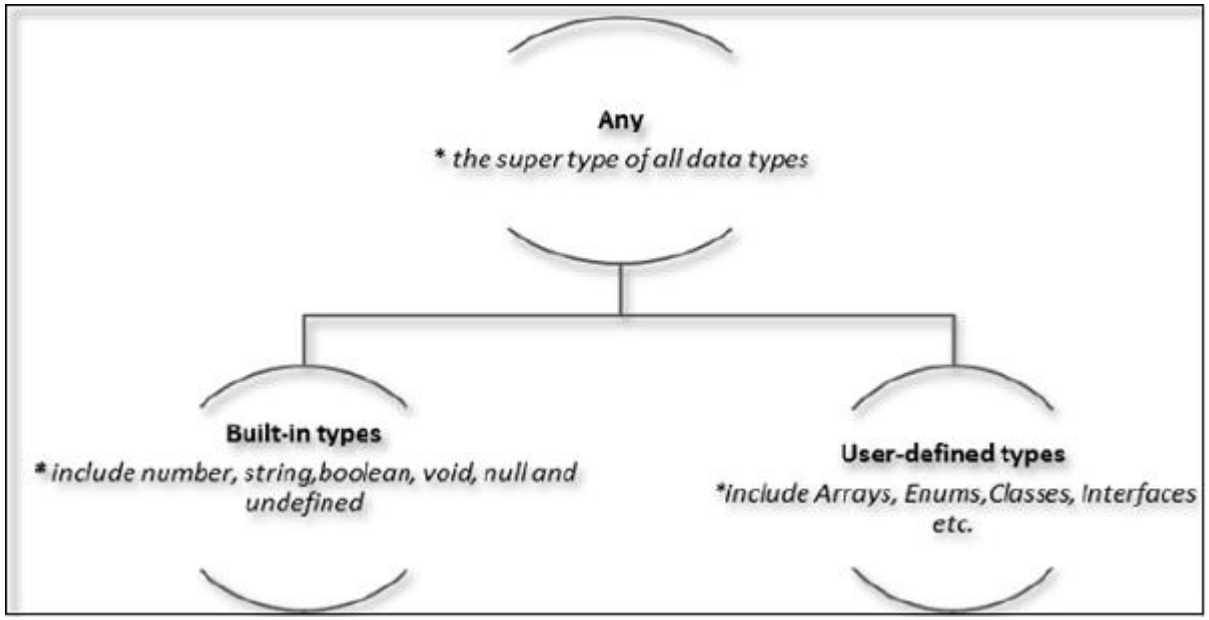
```
for (var i = 0; i < 2; i++) {
        console.log("i="+i);
        setTimeout(function () {
                console.log(i);
        }, 100);
}
console.log("var i="+i);        //?
```

- let

```
for (let i = 0; i < 2; i++) {
        console.log("i="+i);
        setTimeout(function () {
                console.log(i);
        }, 100);
}
console.log("let i="+i);        //ERROR
```

Data Type

- Data type classification



Data Type

- boolean true or false
- number integer and floating point value
- string text value
  - explicit type

```
let bool : boolean = true;
let num : number = 100;
let text : string = “hello world”;
```

  - implicit type (inference)

```
let bool = true;
let num = 100;
let text = “hello world”;
```

Data Type

- Boolean, Number, String
  - string vs String
    - string is primitive type
    - String is an object wraps the primitive type

```
let str1 = "hello";
let str2 = String("hello");
let str3 = new String("hello");
```

```
let hi : String = "hi";
let hello : string = "hello";
hi = hello;              //OK
hello = hi;              //ERROR
```

Data Type

- string vs String

```
let str1 = "hello";
let str2 = String("hello");
let str3 = new String("hello");
```

```
console.log(str1==str2);        //true
console.log(str1==str3);        //true
console.log(str2==str3);        //true
console.log(str1===str2);       //true
console.log(str1===str3);       //false
console.log(str2===str3);       //false
console.log(typeof str1);       //string
console.log(typeof str2);       //string
console.log(typeof str3);       //object
console.log(typeof str1 === 'string');    //true
console.log(typeof str2 === 'string');    //true
console.log(typeof str3 === 'string');    //false
console.log(str1 instanceof String);      //ERROR
console.log(str2 instanceof String);      //ERROR
console.log(str3 instanceof String);      //true
```

Data Type

- string vs String

string primitive	String object
The string primitives are used extensively.	The String object are scarcely used.
The string primitives only hold the value.	The String object have the ability to hold the property.
The string are immutable thus are thread safe.	The String object is mutable.
The string primitive has no methods.	The String object has methods.
Cannot create two different literals with the same value.	You can create new objects with the keyword ‘new’.
It is a primitive data type.	Wraps primitive data type to create an object.
Passed by value that is copy of primitive itself is passed.	Passed by reference to the actual data.
When using eval() these are directly treated as source code.	When using eval() these are treaded as a string.

Data Type

- Array

```
let fruits : string[] = [“Apple”, “Orange”, “Banana”];
```

```
let fruits : Array<string> = [“Apple”, “Orange”, “Banana”];
```

- Tuple

```
let user: [number, string] = [100, "Steve"];
console.log(user[0]); //access

user.push(200, "Smith"); //add
console.log(user); //[ 100, 'Steve', 200, 'Smith' ]
```

Data Type

- Enum

```
enum {
        Sunday,  Monday,  Tuesday, Wednesday, Thursday,
        Friday, Saturday
};
```

- Union
  - syntax (type1 | type2 | type3 | .. | typeN)

```
let code: (number | string);
code = 123;
code = "abc";
function displayCode(code : number | string) {
        console.log(code);
}
```

Casting Type

- as keyword to cast types
  - change the type of the given variable but doesn't change the type of the data

```
let x: unknown = 'hello';
console.log((x as string).length);
```

- <> casting

```
let x: unknown = 'hello';
console.log((<string>x).length);
```

Special Type

- TypeScript has special type that may not refer to any specific type of data
- any is a type that disable type checking

```
let b = true;
b = "string";
//Error: Type 'string' is not assignable to type 'boolean'
```

```
let b : any = true;
b = "string";
//no error as it can be 'any' type
```

Special Type

- unknown is a similar but safer alternative to any

```
let b :  unknown = true;
b = “string”;
//no error
```

Special Type

- any vs unknown
  - any allows being assigned to any type and calling any method while unknown doesn't

```
const a : any = "a";                    //OK
const b : unknown = "b";                //OK

const v1 : string = a;                  //OK
const v2 : string = b;                  /ERROR
const v3 : string = b as string;        //OK

a.trim();                               //OK
b.trim();                               //ERROR
```

Special Type

- void
  - void is used where there is no data

```
function sayHi(): void {
    console.log('Hi!')
}
```

```
let speech: void = sayHi();
console.log(speech);              //Output: undefined
```

Special Type

- never
  - type contains no value
  - type represents function throws an error or contains an indefinite loop

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

```
function reject() {
   return raiseError('Rejected');
}
```

Special Type

- never

```
function checking(a: string | number): boolean {
  if (typeof a === "string") {
    return true;
  } else if (typeof a === "number") {
    return false;
  }
  // make the function valid
  return neverOccur();
}
```

```
function neverOccur(): never {
   throw new Error('Never!');
}
```

Special Type

- void vs never
  - void type can have undefined or null value
  - never cannot have any value

```
let something: void = null;
let nothing: never = null;        //ERROR
```

Special Type

- undefined & null
  - represent no value or absence of any value

```
let y: undefined = undefined;
let z: null = null;
```

  - falsy

```
let a = undefined;
let b = null;

if (!a) console.log('false');        //false
if (!b) console.log('false');        //false
```

Special Type

- undefined & null
  - comparing

```
console.log(null == undefined)      //true
console.log(null === undefined)      //false
```

  - Arithmetic operation

```
let a = 10;
let b: any = undefined;
console.log(a+b);       //NaN
```

```
let a = 10;
let b: any = null;
console.log(a+b);       //10
```

Special Type

- undefined & null

Null	Undefined
Null is the intentional absence of a value (null is explicit)	Undefined is the unintentional absence of a value (undefined is implicit)
Null must be assigned to a variable	The default value of any unassigned variable is undefined.
The typeof null is an object. (and not type null)	Typeof undefined is undefined type
You can empty a variable by setting it to null	You can Undefine a variable by setting it to Undefined
null is always falsy	undefined is always falsy
null is equal to undefined when compared with == (equality check)	
null is not equal to undefined when compared with === (strict equality check)	
When we convert null to a number it becomes zero	when we convert undefined to number it becomes NaN
null is a valid value in JSON.	You can represent undefined as a JSON (JavaScript Object Notation)

Special Type

- Falsy

Value	Description
false	The keyword false.
0	The Number zero (so, also 0.0, etc., and 0x0).
-0	The Number negative zero (so, also -0.0, etc., and -0x0).
0n	The BigInt zero (so, also 0x0n). Note that there is no BigInt negative zero — the negation of 0n is 0n.
"", '', ``	Empty string value.
null	null — the absence of any value.
undefined	undefined — the primitive value.
NaN	NaN — not a number.

Defining Type

- Type Alias

```
type CarYear = number;
type CarType = string;
type CarModel = string;
type Car = {
        year: CarYear,
        type: CarType,
        model: CarModel
}
```

```
type Sedan = Car & {
        gear: string
}
```

```
const carYear: CarYear = 2001;
const car: Car = {
        year: carYear,
        type: "Toyota",
        model: "Corolla"
}
```

```
const sedan : Sedan = {
        year: 2001,
        type: "Toyota",
        model: "Corolla",
        gear: "auto"
};
```

Defining Type

- Object Type

```
const car: { type: string, model: string, year:
number } = {
  type: "Toyota",
  model: "Corolla",
  year: 2009
};
```

  - Optional (?)

```
const car: { type: string, model: string, year?:
number } = {
  type: "Toyota",
  model: "Corolla"
};
car.year = 2009;
```

Defining Type

- Destructuring

```
let car : Car = {
  year: 2001, type: "Toyota",
  options: {
              color: "gray",
              airbag: true
  }
};
```

```
let year = car.year;
let type = car.type;
let color = car.options.color;
let airbag = car.options.airbag;
console.log(year+","+type);
console.log(color+","+airbag);
```

```
let {
              year,
              type,
              options: { color, airbag }
} = car;
console.log(year+","+type);
console.log(color+","+airbag);
```

```
let {
              year: carYear,
              type: carType,
              options: { color: carColor,
              airbag: carAirbag }
} = car;
console.log(carYear+","+carType);
console.log(carColor+","+carAirbag);
```

Defining Type

- Function
  - functions can be of two types: named and anonymous

```
function display() {
        console.log("Hello TypeScript!");
}
```

```
const greeting = function() {
        console.log("Hello TypeScript!");
};
```

```
const greeting = () => {
        console.log("Hello TypeScript!");
};
```

Defining Type

- Function Parameter

```
function Greet1(name: string, greeting: string ) :
string {
        return greeting + ' ' + name + '!';
}
```

  - Optional parameter

```
function Greet2(name: string, greeting?: string ) :
string {
        return greeting + ' ' + name + '!';
}
```

Defining Type

- Function Parameter
  - Default parameter

```
function Greet3(name: string, greeting: string =
"Hello") : string {
        return greeting + ' ' + name + '!';
}
```

```
Greet1("John"); //ERROR An argument for 'greeting' was not provided.
Greet2("Jane"); //undefined Jane!
Greet3("Jack"); //Hello Jack!
```

Defining Type

- Function Parameter
  - Rest parameter

```
function Greet4(greeting: string, ...names: string[])
: string {
        return greeting + " " + names.join(", ") + "!";
}
```

```
Greet4("Hello","John","Jane","Jack"); //Hello John, Jane, Jack!
Greet4("Hello"); //Hello !
```

Defining Type

- Function Parameter
  - Named parameter - I

```
interface GreetParameter {
        greeting?: string,
        name?: string
}
```

```
function Greet5(options: GreetParameter) : string {
        let greeting = options.greeting || "Greet";
        let name = options.name || "";
        return greeting + " " + name;
}
console.log("Greet5",Greet5({}));
```

Defining Type

- Function Parameter
  - Named parameter - II

```
function Greet6({greeting = "Greet", name = ""} :
GreetParameter) : string {
        return greeting + " " + name;
}
console.log("Greet6",Greet6({}));
```

```
function Greet7({greeting = "Greet", name= ""} :
GreetParameter = {}) : string {
        return greeting + " " + name;
}
console.log("Greet7",Greet7());
```

Defining Type

- Function Overload
  - function with the same name with difference parameter type and return type

```
function add(a: string, b: string): string;
function add(a: number, b: number): number;
function add(a: any, b: any): any {
        return a + b;
}
```

```
add("Hello ", "Steve"); // returns "Hello Steve"
add(10, 20); // returns 30
```

Defining Type

- Function Overload
  - overloading with different number of parameters and types with same name is not supported

```
function display(a: string, b: string):void {
        console.log(a + b);
}

function display(a: number): void {
        console.log(a);
}

//ERROR: Duplicate function implementation
```

Defining Type

- Class

```
class People {
        name: string;
}
```

```
const p = new People();
p.name = "Jane";
```

Defining Type

- Class Constructor

```
class People {
        id: number;
        name: string;
        constructor(id: number, name: string) {
                this.id = id;
                this.name = name;
        }
}
```

```
let p = new People(100,"Jane");
```

Defining Type

- Class Constructor

```
class People {
        constructor(id: number,  name: string) {
        }
        public getId() : number {
                return this.id;
        }
}
```

```
let p = new People(100,"Jane");
p.getId();
p.name; //this must public in constructor
```

Defining Type

- Class Constructor

```
class People {
        constructor(private id: number, public name: string) {
        }
        public getId() : number {
                return this.id;
        }
}
```

```
let p = new People(100,"Jane");
p.getId();
p.name;    //NO ERROR: cause public modifier in constructor
```

Defining Type

- Class Members

```
class People {
        id: number;
        public name: string;
        protected credit : number;
        private account: string;
        constructor(id: number, name: string, credit: number,
account: string) {
                this.id = id;
                this.name = name;
                this.credit = credit;
                this.account = account;
        }
        public getCredit() : number { return this.credit; }
        public getAccount() : string { return this.account; }
}
```

Defining Type

- Class Members

```
const p = new People(100,"Jane",1000,"1-1-0001-1");

console.log(p.id);     //OK
console.log(p.name);  //OK
console.log(p.account);  //Property 'account' is private and only
accessible within class 'People'.
console.log(p.credit);  //Property 'credit' is protected and only
accessible within class 'People' and its subclasses.
console.log(p.getAccount());  //OK
console.log(p.getCredit());  //OK
```

Defining Type

- Abstract Class

```
abstract class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    display(): void {
        console.log(this.name);
    }
    abstract find(name: string): Person;
}
```

Defining Type

- Abstract Class

```
class People extends Person {
        id: number;
        protected credit : number;
        private account: string;
        constructor(id: number, name: string, credit: number, account:
string) {
                super(name);
                this.id = id; this.credit = credit; this.account = account;
        }
        public getCredit() : number { return this.credit; }
        public getName() : string { return this.name; }
        public find(name: string) : Person {
                return new People(100,"Test",1000,"0-0-000-0");
        }
}
const p = new People(100,"Jane",1000,"1-1-0001-1");
```

Defining Type

- Interfaces

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

```
const user : User = {
        id: 100,
        name: "John"
};
```

Defining Type

- Interface

```
class People extends Person implements User {
        id: number;
        protected credit : number;
        private account: string;
        constructor(id: number, name: string, credit: number, account:
string) {
                super(name);
                this.id = id; this.credit = credit; this.account = account;
        }
        public getCredit() : number { return this.credit; }
        public getName() : string { return this.name; }
        public find(name: string) : Person {
                return new People(100,"Test",1000,"0-0-000-0");
        }
}
```

Defining Type

- Casting type

```
function getUser() {
    let username : string = "Jane";
    return {
        id : 100,
        name : username
    };
}

let user  = getUser() as User;
```

```
let user : User = {} as User;

let userArray : User[] = [] as User[];
```

Defining Type

- Override

```
class ThaiPeople extends People {
    public display(): void {
        console.log(this.name+" is Thai");
    }
}
```

```
class ThaiPeople extends People {
    public override display(): void {
        console.log(this.name+" is Thai");
    }
}
//key word override in TypeScript 4.3
```

```
const tp = new ThaiPeople(200,"Yim",2000,"2-2-0002-2");
tp.display(); //Yim is Thai
```

Defining Type

- readonly & static modifier
  - readonly is used to make a property as read-only.

```
class Account {
    readonly code: number;
    account: string;
    constructor(code: number, account: string)     {
        this.code = code;
        this.account = account;
    }
}

let ac = new Account(100, "John");
ac.code = 20; //Compiler Error
ac.account = 'Bill';
```

Defining Type

- readonly & static modifier
  - static members of a class are accessed using the class name and dot notation, without creating an object

```
class Circle {
    static pi: number = 3.14;

    static calculateArea(radius:number) {
        return this.pi * radius * radius;
    }
}

Circle.pi;
Circle.calculateArea(5);
```

Utility Type

- Required
  - Required changes all the properties in object to be required

```
interface Car {
        year: number;
        type: string;
        model: string;
        gear?: string;
}
```

```
let car : Required<Car> = {
        year: 2001,
        type: "Totota",
        model: "Corolla",
        gear: "auto"
};
```

Utility Type

- Record
  - Record is a shortcut to defining an object type with specific key type and value type

```
const carModel : Record<string,number> = {
                "Corolla" : 2001,
                "Vios" : 2002
};
```

Utility Type

- Partial
  - Partial changes all the properties in object to be optional

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

```
let user : Partial<User> = { };
user.id = 100;
```

Utility Type

- Omit
  - Omit remove keys from an object type

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

```
const user : Omit<User, "id" | "credit"> = {
        name: "John"
};
```

Utility Type

- Pick
  - Pick removes all but specified keys from an object type

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

```
let user : Pick<User, "name" | "credit"> = {
        name: "Jane",
        credit: 1000
};
```

Utility Type

- Exclude
  - Exclude removes types from a union

```
type CarGear = string | number | boolean;
const gear : Exclude<CarGear, number | boolean> = "auto";
```

```
const gear : Exclude<CarGear, number | boolean> = true;
//Type 'boolean' is not assignable to type 'string'.
```

Utility Type

- ReturnType
  - ReturnType extracts the return type of a function type

```
function createUser() {
  return {
    id: 100,
    name: "John",
    position: "Developer",
    createdAt: new Date()
  };
};
```

```
type UserInfo = ReturnType<typeof
createUser>;
const uinfo : UserInfo = {
                id: 200,
                name: "Jack",
                position: "Programmer",
                createdAt: new Date()
};
```

Utility Type

- Readonly
  - Constructs a type with all properties of type set to readonly
- NonNullable
  - Constructs a type by excluding null and undefined from type
- Parameters
  - Constructs a tuple type from the types used in the parameters of a function type

Utility Type

- Array
  - Array is a special variable, which can hold more than one value
  - Array are resizable and can contain a mix of different data types
  - Array are not associative arrays and must be accessed using nonnegative integer as index

Utility Type

- Array
  - Iteration of Array
    - forEach - a callback function once for each array element
    - find - return the value of the first array element that passes a test function
    - findIndex - returns the index of the first array element that passes a test function
    - map - creates a new array by performing a function on each array element
    - reduce - run a function on each array element to produce a single value
    - filter - creates a new array with array elements that pass a test
    - some - checks if some array values pass a test
    - every - checks if all array values pass a test

Utility Type

- Array
  - Iteration of Array

```
const a = new Array<number>(10,20,30,40);

a.forEach(e => console.log(e));

console.log("find",a.find(e => e > 20));
console.log("findIndex",a.findIndex(e => e > 20));
console.log("map",a.map(e => e + 10));
console.log("reduce",a.reduce((e,v) => e + v,10));
console.log("filter",a.filter(e => e > 20));
console.log("some",a.some(e => e > 20));
console.log("every",a.every(e => e > 20));
```

Utility Type

- Set
  - Set is allows us to store distinct data into list

```
const s = new Set();
s.add("hello");
s.add("new").add("world");
console.log("size",s.size);
console.log("has",s.has("hello"));
console.log("set",s);

const set = new Set<string>(["Hello", "World"]);
console.log("set",set);
```

Utility Type

- Set

Methods	Descriptions
set.add(value)	It is used to add values in the set.
set.has(value)	It returns true if the value is present in the set. Otherwise, it returns false.
set.delete()	It is used to remove the entries from the set.
set.size()	It is used to returns the size of the set.
set.clear()	It removes everything from the set.

Utility Type

- Set
  - Iteration on Set

```
const s = new Set();
s.add("hello").add("new").add("world");
s.add("hello").add("new").add("world");
s.add("Hello");

for(let key of s) {
    console.log(key);
}

s.forEach(key => console.log(key));

for(let it = s.values(), val = null; val = it.next().value;) {
    console.log(val);
}
```

Utility Type

- Map
  - Map is allow us to store data in a key-value pair and remembers the original insertion order of the keys

```
const m = new Map();
m.set("A","Hello");
m.set("B","World");
console.log("A",m.get("A"));

const map = new Map<string,string> ([
            ["A","Hello"],
            ["B","World"]
]);
console.log("map",map);
```

Utility Type

- Map

Methods	Descriptions
map.set(key, value)	It is used to add entries in the map.
map.get(key)	It is used to retrieve entries from the map. It returns undefined if the key does not exist in the map.
map.has(key)	It returns true if the key is present in the map. Otherwise, it returns false.
map.delete(key)	It is used to remove the entries by the key.
map.size()	It is used to returns the size of the map.
map.clear()	It removes everything from the map.

Utility Type

- Map
  - Iteration on Map

```
const m = new Map<string,string>();
m.set("A","Hello");
m.set("B","World");

m.forEach((value,key) => console.log(key+"="+value));

for(let key of m.keys()) { console.log(key+"="+m.get(key)); }

for(let value of m.values()) { console.log(value); }

for(let [key,value] of m.entries()) { console.log(key+"="+value); }

for(let it = m.entries(), val = null; val = it.next().value;) {
        let [key,value] = val; console.log(key,value);
}
```

Utility Type

- Symbol
  - Symbol is a primitive data type and every symbol is unique

```
const sym1 = Symbol();
const sym2 = Symbol();
const sym3 = Symbol("Symbol");

console.log("sym1",sym1); //Symbol()
console.log("sym3",sym3); //Symbol(Symbol)
//console.log(sym1==sym2); //ERROR
```

Utility Type

- Symbol

```
const symColor = Symbol("color");
const symPrint = Symbol("print");
const car = {
        year: 2001,
        type: "Toyota",
        [symColor]: "Black",
        [symPrint]: function() {
                console.log(this.type+" color "+this[symColor]);
        }
};
console.log("car",JSON.stringify(car)); //{"year":2001,"type":"Toyota"}
console.log("color",car[symColor]);
//console.log("color",car["color"]); //ERROR
car[symPrint]();
```

Type Operator

- keyof
  - keyof operator takes an object type and produces a string or numeric literal union of it's key

```
type CarProperty = keyof Car;
//year | type | model

function printCarProperty(car: Car, property: keyof Car) {
        console.log(`car property ${property}: "${car[property]}"`);
}

printCarProperty(car,"type");
printCarProperty(car,"model");
```

Type Operator

- keyof

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

```
let car: Car = {
  year: 2001,
  type: "Toyota"
};
```

```
printCarProperty(car,"type");  //car property type: "Toyota"
printCarProperty(car,"model"); //car property model: "undefined"
```

Type Operator

- ?
  - Optional chaining use in optional property access and optional call

```
function getCarModel(car?: Car) {
            return car?.model;
}

let car1 : Car = {
  year: 2001, type: "Toyota", model: "Corolla"
};

let car2 : Car = undefined;
console.log(getCarModel(car1));  //Corolla
console.log(getCarModel(car2));  //undefined
```

Type Operator

- ??
  - Nullish coalescence is a logical operator that return its right-hand side operand when its left-hand side operand is null or undefined

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

```
let car: Car = {
  year: 2001,
  type: "Toyota"
};
```

```
let carModel = car.model??"Vios";
console.log(carModel);  //Vios
```

```
let carModel = car.model !== null && car.model !== undefined ?
car.model : "Vios";
```

Type Operator

- ...
  - the spread syntax to merge objects

```
let car1 : Car = {
  year: 2001, type: "Toyota", model: "Corolla"
};

let car2 = {...car1, gear:"auto"};
console.log(car2);
let carOptions = {color: "gray", airbag: true};
let car3 = {...car1, ...carOptions};
console.log(car3);
let car4 = {...car1, model:"Corona"};
console.log(car4);
let car5 : Car = undefined;
let car6 = { ...car1, ...car5 };
console.log(car6)
```

Type Operator

- Spread vs Object.assign
  - Shallow copied
  - Object.assign target can be mutated

```
let car0 = { year: 2001, type: "Toyota", model: "Corolla" };
let car1 = { year: 2005, type: "Toyota", options: {color: "gray", airbag:
true} };
let car2 = {...car1};
let car3 = Object.assign(car0,car1);
car2.year = 2009;
car3.year = 2010;
car3.options.color = "black";
console.log("car0",car0);
console.log("car1",car1);
console.log("car2",car2);
console.log("car3",car3);
```

Type Operator

- =>
  - arrow function

```
var greeting = function(name) {
  console.log('Hello ' + name);
}

var add = function(a, b) {
  return a + b;
}
```

```
var greeting = name => {
  console.log('Hello ' + name);
}

var add = (a, b) => a + b;
```

Type Operator

- =>
  - arrow function

```
function Greeting() {
    this.name = 'Jack',
    this.sayLater = function () {
        console.log(this.name);
        setTimeout(function() {
            console.log(this.name);
        }, 1000);
    }
}
let greet = new Greeting();
greet.sayLater();
```

```
function Greeting() {
    this.name = 'Jack',
    this.sayLater = function () {
        console.log(this.name);
        setTimeout(() => {
            console.log(this.name);
        }, 1000);
    }
}
let greet = new Greeting();
greet.sayLater();
```

Type Operator

- Template Strings
  - grave accent (`…`) [Alt+96]

```
let car : Car = {
  year: 2001, type: "Toyota",
  options: {
              color: "gray",
              airbag: true
  }
};

console.log(`My car : year=${car.year}, type=${car.type}`);
console.log(`My car : color=${car.options.color}`);
```

Type Operator

- async & await
  - async is a function declared enable asynchronous or promise based behavior
  - await operator is used to wait for a Promise

```
async function name(param1,param2,paramN) {
        statements
}
```

```
[return value] = await expression
```

Type Operator

- async & await
  - callback function

```
function doTaskA(param: number, callback: Function) {
    callback("A",param+1);
}
function doTaskB(param: number, callback: Function) {
    callback("B",param+2);
}
function doTaskC(param: number, callback: Function) {
    callback("C",param+3);
}
```

Type Operator

- async & await
  - callback function

```
function doMyTask() {
    doTaskA(10,function(aName: string,aValue: number)
{
        console.log(aName,aValue);
        doTaskB(20,function(bName: string, bValue:
number) {
            console.log(bName,bValue);
            doTaskC(30,function(cName: string, cValue:
number) {
                console.log(cName,cValue);
            });
        });
    });
}
```

Type Operator

- async & await
  - Promise object represents the eventual completion or failure of an asynchronous operation

Type Operator

- async & await
  - Promise

```
interface Task { name: string, value: number };
function doPromA(param: number) : Promise<Task> {
    return new Promise((resolve,reject) => {
        resolve({name: "A", value: param+1});
    });
}
function doPromB(param: number) : Promise<Task> {
    return new Promise((resolve,reject) => {
        resolve({name: "B", value: param+2});
    });
}
function doPromC(param: number) : Promise<Task> {
    return new Promise((resolve,reject) => {
        resolve({name: "C", value: param+3});
    });
}
```

Type Operator

- async & await
  - Promise

```
function doMyProm() {
    doPromA(10).then(taskA => {
        console.log(taskA);
        doPromB(20).then(taskB => {
            console.log(taskB);
            doPromC(30).then(taskC => {
                console.log(taskC);
            });
        });
    });
}
```

Type Operator

- async & await
  - Promise

```
function doMyProcess() {
    doPromA(10).then(taskA => {
        console.log(taskA);
        return doPromB(20);
    }).then(taskB => {
        console.log(taskB);
        return doPromC(30);
    }).then(taskC => {
        console.log(taskC);
    });
}
```

Type Operator

- async & await
  - Promise

```
async function doMyAsync() {
    let taskA = await doPromA(10);
    let taskB = await doPromB(20);
    let taskC = await doPromC(30);
    console.log(taskA);
    console.log(taskB);
    console.log(taskC);
}
```

Type Operator

- async & await
  - Promise
    - parallel task

```
async function doParallel() {
    let taskA = await doPromA(10);
    let [taskB, taskC] = await
Promise.all([doPromB(20), doPromC(30)]);
    console.log(taskA);
    console.log(taskB);
    console.log(taskC);
}
```

Type Operator

- async & await
  - Handle exception

```
async function doMyAsync() {
    try {
        let taskA = await doPromA(10);
        let taskB = await doPromB(20);
        let taskC = await doPromC(30);
        console.log(taskA);
        console.log(taskB);
        console.log(taskC);
    } catch(ex) {
        console.error(ex);
    }
}
```

Type Guard

- A type guard is a function that allows you to narrow the type of an object to a more specific one by performing certain checks

Type Guard

- typeof
  - typeof only returns one of the following: “string”, “number”, “bigint”, “boolean”, “symbol”, “undefined”, “object”, “function”

```
typeof  90 === “number”

typeof  “abc” === “string”
```

Type Guard

- typeof

Type	Predicate
string	typeof s === "string"
number	typeof n === "number"
boolean	typeof b === "boolean"
undefined	typeof undefined === "undefined"
function	typeof f === "function"
array	Array.isArray(a)
object	typeof o === "object"
bigint	typeof m === "bigint"
symbol	typeof g === "symbol"

Type Guard

- instanceof
  - instanceof checks whether an object is of a specific type or not
  - instanceof does not work with interfaces

```
new Date() instanceof Date === true
```

Type Guard

- type predicates
  - A type predicate is a return type of a function to defined as user-defined type guard

```
interface Book {
        id: number;
        author: string;
        publisher?: string;
}
```

Type Guard

- type predicates

```
const isBook = (element: unknown) : element is Book  => {
        return Object.prototype.hasOwnProperty.call(element, "id")
        &&
        Object.prototype.hasOwnProperty.call(element, "author");
}
```

```
const isBook = (element: unknown) : element is Book  => {
        return Object.prototype.hasOwnProperty.call(element, "id")
        && typeof element.id === "number“ &&
        Object.prototype.hasOwnProperty.call(element, "author")
        && typeof element.author === "string";
}
//ERROR
```

Type Guard

- type predicates

```
const isBook = (element: unknown) : element is Book  => {
        return hasAttributes(element,  ["id", "author"]) &&
        typeof element.id === "number" &&
        typeof element.author === "string";
}
```

```
const hasAttributes =  <T extends string> (
        element: unknown,
        attributes: T[]
) : element is Record<T, unknown>  => {
        if(element === undefined || element === null) return false;
        return attributes.every((attribute) =>
                return
Object.prototype.hasOwnProperty.call(element, attribute);
        );
}
```

Module

- Module is a way to create a group of related variables, functions, classes and interfaces
  - Internal module
  - External module

Module

- Internal Module
  - Logical grouping of classes, interfaces, functions, variables into a single unit and can be exported to another module

```
export interface Book {
        id: number;
        author: string;
        publisher?: string;
}
export const isBook = (element: unknown) : element is Book  => {
        return hasAttributes(element,  ["id", "author"]) &&
        typeof element.id === "number" &&
        typeof element.author === "string";
}
```

Module

- External Module
  - Known as a module is used to specify the load dependencies between the multiple external java script files

```
import { Book, isBook } from "./library";

let b1 : Book = { id: 100, author: "Stephen King" };
let b2 = { id: 100, publisher: "The Shining" };
let b3 = { id: 300, author: "Agatha Christie" };

console.log("b1 is book",isBook(b1));
console.log("b2 is book",isBook(b2));
console.log("b3 is book",isBook(b3));
```

Module

- namespace
  - Brand-new of module to organize code

```
export namespace Shapes {
        export interface Shape {
                area(): number;
        }
        export class Triangle implements Shape {
                constructor(private w: number, private h: number) {}
                public area() : number { return 0.5*this.w*this.h; }
        }
        export class Square implements Shape {
                constructor(private w: number, private h: number) {
}
                public area() : number { return this.w*this.h; }
        }
}
```

Module

- namespace

```
import { Shapes } from "./shapes";

let s1 : Shapes.Shape = new Shapes.Triangle(10,20);
let s2 = new Shapes.Square(10,20);
console.log(s1.area());
console.log(s2.area());
```

Reference

- https://www.w3schools.com/typescript/index.php
- https://www.typescriptlang.org/docs/
- https://www.tektutorialshub.com/typescript/
- https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html
- https://www.tutorialspoint.com/typescript/typescript_overview.htm
- https://www.tutorialsteacher.com/typescript
- https://www.javatpoint.com/typescript-tutorial

Q & A