Typescript

TypeScript stands in an unusual relationship to JavaScript. TypeScript offers all of JavaScript's features, and an additional layer on top of these: TypeScript's type system.

For example, JavaScript provides language primitives like `string` and `number`, but it doesn't check that you've consistently assigned these. TypeScript does.

This means that your existing working JavaScript code is also TypeScript code. The main benefit of TypeScript is that it can highlight unexpected behavior in your code, lowering the chance of bugs.

Install TypeScript

To install or update the latest version of TypeScript, open command prompt/terminal and type the following command:

```
npm install -g typescript
```

The above command will install TypeScript globally so that you can use it in any project. Check the installed version of TypeScript using the following command:

```
tsc -v
```

Execute the following command to install the TypeScript to your local project as dev dependency.

```
npm install typescript --save-dev
```

Run Follwoing command to compile .ts files

```
tsc filename.ts
```

```
tsc my-first.ts
```

Run watch

```
tsc -w filename.ts
```

Run Compiled JS file in terminal to view output

```
node filename.js
```

What is tsconfig.json?

TypeScript Playground

TypeScript provides an online playground https://www.typescriptlang.org/play to write and test your code on the fly without the need to download or install anything.

Type Annotations

When you declare a variable using `const`, `var`, or `let`, you can optionally add a type annotation to explicitly specify the type of the variable: TypeScript includes all the primitive types of JavaScript- number, string and boolean.

```
var age: number = 32; // number variable
var name: string = "John";// string variable
var isUpdated: boolean = true;// Boolean variable
```

In the above example, each variable is declared with their data type. These are type annotations. You cannot change the value using a different data type other than the declared data type of a variable. If you try to do so, TypeScript compiler will show an error. This helps in catching JavaScript errors. For example, if you assign a string to a variable age or a number to name in the above example, then it will give an error.

- **Type Annotation of Parameters**

```
function display(id:number, name:string)
{
    console.log("Id = " + id + ", Name = " + name);
}
```

- **Type Annotation in Object**

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

employee = {
  id: 100,
  name : "John"
}
```

- **Return Type Annotations**

```
function getFavoriteNumber(): number {
  return 26;
}
```

Basic Types

- Number

```
let first:number = 123; // number
let second: number = 0x37CF;  // hexadecimal
let third:number=0o377 ;      // octal
let fourth: number = 0b111001;// binary

console.log(first);  // 123
console.log(second); // 14287
console.log(third);  // 255
console.log(fourth); // 57
```

  - **String**

```
let color: string = "blue";
color = 'red';
```

  - **Boolean**
    - `let isPresent:boolean = true;`
  - **Arrays**
    - To specify the type of an array like `[1, 2, 3]`, you can use the syntax `number[]`; this syntax works for any type (e.g. `string[]` is an array of strings, and so on). You may also see this written as `Array<number>`, which means the same thing. We'll learn more about the syntax `T<U>` when we cover *generics*.
    - Declare array using square methods
      - `let fruits: string[] = ['Apple', 'Orange', 'Banana'];`

- Using a generic array type, Array<elementType>.
  - `let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];`
- Multi Type Array

```
let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
// or
let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
```

- **Tuple**
  - Tuple types allow you to express an array with a fixed number of elements whose types are known, but need not be the same. For example, you may want to represent a value as a pair of a `string` and a `number`:
  -

```
var employee: [number, string] = [1, "Mukesh"];
var person: [number, string, boolean] = [1, "Mukesh", true];

var user: [number, string, boolean, number, string];// declare tuple variable
user = [1, "Mukesh", true, 20, "Admin"];// initialize tuple variable
```

  - Tuple Array

```
var employee: [number, string][];
employee = [[1, "Mukesh"], [2, "Bill"], [3, "Jeff"]];
```

- **Enum**
  - Enums or enumerations are a new data type supported in TypeScript. Most object-oriented languages like Java and C# use enums. This is now available in TypeScript too.
  - Numeric Enum
    -

```
enum PrintMedia {
  Newspaper,
  Newsletter,
  Magazine,
  Book
}
```

      In the above example, we have an enum named PrintMedia. The enum has four values: Newspaper, Newsletter, Magazine, and Book. Here, enum values start from zero and increment by 1 for each member. It would be represented as:

```
Newspaper = 0
Newsletter = 1
Magazine = 2
Book = 3
```

    - We can assign values to enum members
      -

```
enum PrintMedia {
    Newspaper = 1,
    Newsletter = 5,
    Magazine = 15,
    Book = 10
}
//Accessing enums
PrintMedia.Newsetter; // returns 5
PrintMedia.Magazine; // returns 15
```

  - String Enum

```
enum PrintMedia {
    Newspaper = "NEWSPAPER",
    Newsletter = "NEWSLETTER",
    Magazine = "MAGAZINE",
    Book = "BOOK"
}
// Access String Enum
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE
```

- Heterogeneous Enum

```
enum Status {
    Active = 'ACTIVE',
    Deactivate = 1,
    Pending
}
```

- **Union**

```
let code: (string | number);
code = 123;   // OK
code = "ABC"; // OK
code = false; // Compiler Error

let empId: string | number;
empId = 111; // OK
empId = "E111"; // OK
empId = true; // Compiler Error
```

- **Any**
  - TypeScript also has a special type, `any`, that you can use whenever you don't want a particular value to cause typechecking errors.

    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 obj: any = { x: 0 };// None of the following lines of code will throw compiler errors.//
Using `any` disables all further type checking, and it is assumed
// you know the environment better than TypeScript.
obj.foo();
obj();
obj.bar = 100;
obj = "hello";
const n: number = obj;
```

- **Void**
  - `void` is a little like the opposite of `any`: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:
  -

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

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

- **Never**
  - TypeScript introduced a new type `never`, which indicates the values that will never occur.

    The never type is used when you are sure that something is never going to occur. For example, you write a function which will not return to its end point or always throws an exception.

  - ```
    function throwError(errorMsg: string): never {
                throw new Error(errorMsg);
    }

    function keepProcessing(): never {
                while (true) {
              console.log('I always does something and never ends.')
          }
    }
    ```

- **Object**
  - `object` is a type that represents the non-primitive type, i.e. anything that is not `number`, `string`, `boolean`, `bigint`, `symbol`, `null`, or `undefined`.
  - ```
    function create(o: object | null): void;
    create({ prop: 0 });
    ```

- **Unknown**
  - We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content – e.g. from the user – or we may want to intentionally accept all values in our API. In these cases, we want to provide a type that tells the compiler and future readers that this variable could be anything, so we give it the `unknown` type.

    ```
    let notSure: unknown = 4;notSure = "maybe a string instead";
    // OK, definitely a booleannotSure = false;
    ```

Type Inference

TypeScript is a typed language. However, it is not mandatory to specify the type of a variable. TypeScript infers types of variables when there is no explicit information available in the form of type annotations. Types are inferred by the TypeScript compiler when **Variables are initialized, Default values set to parameters, and returning the value from the function. For e.g**

`var a = "some text"`

Here, since we are not explicitly defining `a: string` with a type annotation, TypeScript infers the type of the variable based on the value assigned to the variable.

```
var b = 123;
a = b; // Compiler Error: Type 'number' is not assignable to type 'string'
```

The above code shows an error because while inferring types, TypeScript inferred the type of variable `a` as string and variable `b` as number. When we try to assign `b` to `a`, the compiler complains saying a number type cannot be assigned to a string type.

- **Type inference in complex objects**
  - ```
    var arr = [0, 1, "test"];
    arr.push("str") // OK
    arr.push(true); // Compiler Error: Argument of type 'true' is not assignable to parameter of type 'string
    | number'
    ```
    ```

Type Assertion

```
let code: any = 123;
let employeeCode = <number> code;
console.log(typeof(employeeCode)); //Output: number
```

In the above example, we have a variable `code` of type `any`. We assign the value of this variable to another variable called `employeeCode`. However, we know that code is of type number, even though it has been declared as 'any'. So, while assigning `code` to `employeeCode`, we have asserted that code is of type number in this case, and we are certain about it. Now, the type of `employeeCode` is number.

- **Type Assertion with Object**
  - ```
    let 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 '{}'
    ```
  - ```
    interface Employee {
        name: string;
        code: number;
    }

    let employee = <Employee> { };
    employee.name = "John"; // OK
    employee.code = 123; // OK
    ```
- **Type Assertion using as syntax**
  - ```
    let code: any = 123;
    let employeeCode = code as number;
    ```

Functions

Functions ensure that the program is maintainable and reusable, and organized into readable blocks. While TypeScript provides the concept of classes and modules, functions still are an integral part of the language.In TypeScript, functions can be of two types: named and anonymous.

- **Named Functions**
  - A named function is one where you declare and call a function by its given name.
  - ```
    function Sum(x: number, y: number) : number {
        return x + y;
    }

    Sum(2,3); // returns 5
    ```
- **Anonymous Function**
  - ```
    let Sum = function(x: number, y: number) : number
    {
        return x + y;
    }

    Sum(2,3); // returns 5
    ```

- **Typing the function**
    ```

-
```
function add(x: number, y: number): number {
  return x + y;
}

let myAdd = function (x: number, y: number): number {
  return x + y;
};
```

  We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

- **Writing the function type**
  -
```
let myAdd: (x: number, y: number) => number = function (
  x: number,
  y: number
): number {
  return x + y;
};
```

  - The above function type can be written in the interface as follows:

```
interface IFunctionType {
  (x: number, y: number): number
}

//names function type in inteface
interface IFunctionType {
  myAdd(x: number, y: number): number
}
```

- **Optional Parameters**
  - We can model optional parameters in TypeScript by marking the parameter as *optional* with `?`:
  -
```
function f(x?: number) {
  // ...
}
f(); // OK
f(10); // OK
```

- **Default Parameters**
  - In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes `undefined` in its place. These are called default-initialized parameters.
  -
```
function buildName(firstName: string, lastName = "Smith") {
  return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"
let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
//Expected 1-2 arguments, but got 3.
let result4 = buildName("Bob", "Adams"); // ah, just right
let result5 = buildName("Adams", undefined); // okay and returns "Adams smith"
```

- **Rest Parameters**

- When the number of parameters that a function will receive is not known or can vary, we can use rest parameters. In JavaScript, this is achieved with the "arguments" variable. However, with TypeScript, we can use the rest parameter denoted by ellipsis `...`.

  We can pass zero or more arguments to the rest parameter. The compiler will create an array of arguments with the rest parameter name provided by us.

- ```
  function multiply(n: number, ...m: number[]) {
    return m.map((x) => n * x);
  }
  // 'a' gets value [10, 20, 30, 40]
  const a = multiply(10, 1, 2, 3, 4);
  ```

- **Rest Arguments**
  - Conversely, we can *provide* a variable number of arguments from an array using the spread syntax. For example, the `push` method of arrays takes any number of arguments:
  - ```
    const arr1 = [1, 2, 3];
    const arr2 = [4, 5, 6];
    arr1.push(...arr2);
    ```

- **Parameter Destructuring**
  - ```
    function sum({ a, b, c }) {
      console.log(a + b + c);
    }
    sum({ a: 10, b: 3, c: 9 });
    ```
  - The type annotation for the object goes after the destructuring syntax:
    - ```
      function sum({ a, b, c }: { a: number; b: number; c: number }) {
        console.log(a + b + c);
      }
      ```
    - This can look a bit verbose, but you can use a named type here as well:

      ```
      // Same as prior example
      type ABC = { a: number; b: number; c: number }; // We can create custom types
      function sum({ a, b, c }: ABC) {
        console.log(a + b + c);
      }
      ```

- **Arrow Functions**
  - Fat arrow notations are used for anonymous functions i.e for function expressions. They are also called lambda functions in other languages.
  - ```
    let sum = (x: number, y: number): number => {
        return x + y;
    }

    sum(10, 20); //returns 30
    ```
  - Parameterless Arrow Function
    - ```
      let Print = () => console.log("Hello TypeScript");

      Print(); //Output: Hello TypeScript
      ```

- **Function Overloading**

- TypeScript provides the concept of function overloading. You can have multiple functions with the same name but different parameter types and return type. However, the number of parameters should be the same.

```
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
```

Type Aliases

We've been using object types and union types by writing them directly in type annotations. This is convenient, but it's common to want to use the same type more than once and refer to it by a single name. A *type alias* is exactly that - a *name* for any *type*. The syntax for a type alias is:

```
type Point = {
  x: number;
  y: number;
};
function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}
printCoord({ x: 100, y: 100 });
```

Interfaces

An *interface declaration* is another way to name an object type. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface. The interface is defined with the keyword `interface` and it can include properties and method declarations using a function or an arrow function.

```
interface IEmployee {
    empCode: number;
    empName: string;
    getSalary: (number) => number; // arrow function
    getManagerName(number): string; // Normal Function
}
```

- **Interface as Type**
  - Interface in TypeScript can be used to define a type

```
interface KeyPair {
    key: number;
    value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK

let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error: 'val' doesn't exist in type 'KeyPair'

```

```
let kv3: KeyPair = { key:1, value:100 }; // Compiler Error:
```

- **Interface as function type**

```
interface KeyValueProcessor{
    (key: number, value: string): void;
};

function addKeyValue(key:number, value:string):void {
    console.log('addKeyValue: key = ' + key + ', value = ' + value)
}

let kvp: KeyValueProcessor = addKeyValue;
kvp(1, 'Bill'); //Output: addKeyValue: key = 1, value = Bill
```

- **Interface for Array Type**

```
interface IStringList {
    [index:string]:string
}

let strArr : IStringList;
strArr["TS"] = "TypeScript";
strArr["JS"] = "JavaScript";
```

- **Optional Property**

```
interface IEmployee {
    empCode: number;
    empName: string;
    empDept?:string;
}

let empObj1:IEmployee = {   // OK
    empCode:1,
    empName:"Steve"
}

let empObj2:IEmployee = {    // OK
    empCode:1,
    empName:"Bill",
    empDept:"IT"
}
```

- **Read only Properties**
  - TypeScript provides a way to mark a property as read only. This means that once a property is assigned a value, it cannot be changed!

```
interface Citizen {
    name: string;
    readonly SSN: number;
}

let personObj: Citizen  = { SSN: 110555444, name: 'James Bond' }

personObj.name = 'Steve Smith'; // OK
personObj.SSN = '333666888'; // Compiler Error
```

- **Extending Interfaces**

- Interfaces can extend one or more interfaces. This makes writing interfaces flexible and reusable.
- Like classes, interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components.
- ```
  interface IPerson {
      name: string;
      gender: string;
  }

  interface IEmployee extends IPerson {
      empCode: number;
  }

  let empObj:IEmployee = {
      empCode:1,
      name:"Bill",
      gender:"Male"
  }
  ```

- **Implementing an Interface**
  - Similar to languages like Java and C#, interfaces in TypeScript can be implemented with a Class. The Class implementing the interface needs to strictly conform to the structure of the interface.
  - ```
    interface IEmployee {
        empCode: number;
        name: string;
        getSalary:(number)=>number;
    }

    class Employee implements IEmployee {
        empCode: number;
        name: string;

        constructor(code: number, name: string) {
                    this.empCode = code;
                    this.name = name;
        }

        getSalary(empCode:number):number {
            return 20000;
        }
    }

    let emp = new Employee(1, "Steve");
    ```

Classes

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers can build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

```
class Employee {
    empCode: number;
    empName: string;
```

```
4
5       constructor(code: number, name: string) {
6               this.empName = name;
7               this.empCode = code;
8       }
9
10      getSalary() : number {
11          return 10000;
12      }
13  }
```

- **Constructor**
  - The constructor is a special type of method which is called when creating an object. In TypeScript, the constructor method is always defined with the name "constructor".
  - ```
    1  class Employee {
    2
    3      empCode: number;
    4      empName: string;
    5
    6      constructor(empcode: number, name: string ) {
    7              this.empCode = empcode;
    8              this.name = name;
    9      }
    10 }
    11
    12 let emp = new Employee(100,"Steve");
    ```

- **Inheritance**
  - Just like object-oriented languages such as Java and C#, TypeScript classes can be extended to create new classes with inheritance, using the keyword `extends`
  - ```
    1  class Animal {
    2    move() {
    3      console.log("Moving along!");
    4    }
    5  }
    6
    7  class Dog extends Animal {
    8    woof(times: number) {
    9      for (let i = 0; i < times; i++) {
    10       console.log("woof!");
    11     }
    12   }
    13 }
    14
    15 const d = new Dog();
    16 // Base class method
    17 d.move();
    18 // Derived class method
    19 d.woof(3);
    ```
  - **Super Calls**
    - Just as in JavaScript, if you have a base class, you'll need to call `super();` in your constructor body before using any `this.` members:

      ```
      1  class Base {
      2    k = 4;
      ```

```
}
class Derived extends Base {
  constructor() {
    // Prints a wrong value in ES5; throws exception in ES6
    console.log(this.k);
    'super' must be called before accessing 'this' in the constructor of a derived class.'super' must be called before accessing 'this' in the constructor of a derived class.
    super();
  }
}
```

- **Interface extends classes**

```
class Person {
    name: string;
}

interface IEmployee extends Person {
    empCode: number;
}

let emp: IEmployee = { empCode  : 1, name:"James Bond" }
```

- **Class implementations**
  - A class can implement single or multiple interfaces.

```
interface Pingable {
  ping(): void;
}

class Sonar implements Pingable {
  ping() {
    console.log("ping!");
  }
}

class Ball implements Pingable {
Class 'Ball' incorrectly implements interface 'Pingable'.
  Property 'ping' is missing in type 'Ball' but required in type 'Pingable'.
  pong() {
    console.log("pong!");
  }
}
```

- **Getters / Setters**
  - Classes can also have *accessors*:

```
class A {
  private _length = 0;
  get length() {
    return this._length;
  }
  set length(value) {
    this._length = value;
  }
}

let clsA = new A();
```

```
clsA.length = 10;

// If you have a getter without a setter, the field is automatically readonly
// It is not possible to have accessors with different types for getting and setting.
// To implement accessor -> propert must be defined with underscore and outside class we have to use it without underscore
// Private field can be accessible outside the class if getter and setter exists
```

- **Readonly class property**
  - ```
    class Employee {
        readonly empCode: number;
        empName: string;
    }
    let emp = new Employee();
    emp.empCode = 20; //Compiler Error
    ```

- **Data Modifiers**
  - public
    - By default, all members of a class in TypeScript are public. All the public members can be accessed anywhere without any restrictions.
    - ```
      class Employee {
          public empCode: string = "213";
          empName: string = "Mukesh"; // It is by default public
      }

      let emp = new Employee();
      emp.empCode = "5623";
      emp.empName = "Swati";
      ```
  - protected
    - `protected` members are only visible to subclasses of the class they're declared in.
    - ```
      class Greeter {
        public greet() {
          console.log("Hello, " + this.getName());
        }
        protected getName() {
          return "hi";
        }
      }

      class SpecialGreeter extends Greeter {
        public howdy() {
          // OK to access protected member here
          console.log("Howdy, " + this.getName());
        }
      }
      const g = new SpecialGreeter();
      g.greet(); // OK
      g.getName();
      //Property 'getName' is protected and only accessible within class 'Greeter' and its subclasses.
      ```
  - private
    - `private` is like `protected`, but doesn't allow access to the member even from subclasses:
    - ```
      class Base {
        private x = 0;
      ```

```
}
class Derived extends Base {
  //Class 'Derived' incorrectly extends base class 'Base'.
  //Property 'x' is private in type 'Base' but not in type 'Derived'.
  x = 1;
}

const b = new Base();
// Can't access from outside the class
console.log(b.x);
```

- **Static Members**
  - Classes may have `static` members. These members aren't associated with a particular instance of the class. They can be accessed through the class constructor object itself:
  - ```
    class Circle {
        static pi: number = 3.14;

        static calculateArea(radius:number) {
            return this.pi * radius * radius;
        }
    }
    Circle.pi; // returns 3.14
    Circle.calculateArea(5); // returns 78.5
    ```
  - Why no static classes?
- **Abstract Class**
  - Classes, methods, and fields in TypeScript may be *abstract*.

    An *abstract method* or *abstract field* is one that hasn't had an implementation provided. These members must exist inside an *abstract class*, which cannot be directly instantiated.

    The role of abstract classes is to serve as a base class for subclasses which do implement all the abstract members. When a class doesn't have any abstract members, it is said to be *concrete*.
  - ```
    abstract class Person {
        abstract name: string;

        display(): void{
            console.log(this.name);
        }
    }

    class Employee extends Person {
        name: string;
        empCode: number;

        constructor(name: string, code: number) {
            super(); // must call super()

            this.empCode = code;
            this.name = name;
        }
    }

    let personObj = new Person(); // Can not create object of abstract class
    let emp: Person = new Employee("James", 100);
    emp.display(); //James
    ```

Generics

The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the `echo` command.

Without generics, we would either have to give the identity function a specific type:

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

Or, we could describe the identity function using the `any` type:

```
function identity(arg: any): any {
  return arg;
}
```

While using `any` is certainly generic in that it will cause the function to accept any and all types for the type of `arg`, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a *type variable*, a special kind of variable that works on types rather than values.

```
function identity<Type>(arg: Type): Type {
  return arg;
}
```

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
let output = identity<string>("myString");
let output: string
```

E.g

```
function getArray<T>(items : T[] ) : T[] {
    return new Array<T>().concat(items);
}

let myNumArr = getArray<number>([100, 200, 300]);
let myStrArr = getArray<string>(["Hello", "World"]);

myNumArr.push(400); // OK
myStrArr.push("Hello TypeScript"); // OK

myNumArr.push("Hi"); // Compiler Error
myStrArr.push(500); // Compiler Error
```

- **Multiple Type Variables**
  - ```
    function displayType<T, U>(id:T, name:U): void {
      console.log(typeof(id) + ", " + typeof(name));
    }

    displayType<number, string>(1, "Steve"); // number, string
    ```
    ```

- **Generic Type Methods and Properties**
  - ```
    function displayType<T, U>(id:T, name:U): void {

      if(typeof(id) == 'number') {
        id.toFixed();
      }
       if(typeof(id) == 'string') {
        id.toString();
      }

    // Compiler Error: 'toFixed' does not exists on type 'T'
        name.toUpperCase(); // Compiler Error: 'toUpperCase' does not exists on type 'U'

        console.log(typeof(id) + ", " + typeof(name));
    }
    ```

- **Generic Array Methods**
  - ```
    function displayNames<T>(names:T[]): void {
        console.log(names.join(", "));
    }

    displayNames<string>(["Steve", "Bill"]); // Steve, Bill
    ```

- **Generic Constraints**
  - As mentioned above, the generic type allows any data type. However, we can restrict it to certain types using constraints. Consider the following example:
  - ```
    class Person {
        firstName: string;
        lastName: string;

        constructor(fname:string,  lname:string) {
            this.firstName = fname;
            this.lastName = lname;
        }
    }

    function display<T extends Person>(per: T): void {
        console.log(`${ per.firstName} ${per.lastName}` );
    }
    var per = new Person("Bill", "Gates");
    display(per); //Output: Bill Gates

    display("Bill Gates");//Compiler Error
    ```
  - More examples
    - ```
      function loggingIdentity<Type>(arg: Type): Type {
        console.log(arg.length);
      Property 'length' does not exist on type 'Type'.
        return arg;
      }
      ```
    - ```
      interface Lengthwise {
        length: number;
      }

      ```
      ```

```
function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length); // Now we know it has a .length property, so no more error
  return arg;
}
```

- **Generic Interface**

```
interface KeyPair<T, U> {
    key: T;
    value: U;
}

let kv1: KeyPair<number, string> = { key:1, value:"Steve" }; // OK
let kv2: KeyPair<number, number> = { key:1, value:12345 }; // OK
```

- **Generic Classes**

```
interface IKeyValueProcessor<T, U>
{
    process(key: T, val: U): void;
};

class kvProcessor<T, U> implements IKeyValueProcessor<T, U>
{
    process(key:T, val:U):void {
        console.log(`Key = ${key}, val = ${val}`);
    }
}

let proc: IKeyValueProcessor<number, string> = new kvProcessor();
proc.process(1, 'Bill'); //Output: key = 1, value = Bill
```

Namespaces

The namespace is used for logical grouping of functionalities. A namespace can include interfaces, classes, functions and variables to support a single or a group of related functionalities.

```
namespace StringUtility {

    export function ToCapital(str: string): string {
        return str.toUpperCase();
    }

    export function SubString(str: string, from: number, length: number = 0): string {
        return str.substr(from, length);
    }
}
```

Now, we can use the `StringUtility` namespace elsewhere. The following JavaScript code will be generated for the above namespace.

```
/// <reference path="StringUtility.ts" />

export class Employee {
    empCode: number;
    empName: string;
    constructor(name: string, code: number) {
```

```
        this.empName = StringUtility.ToCapital(name);
        this.empCode = code;
    }
    displayEmployee() {
        console.log ("Employee Code: " + this.empCode + ", Employee Name: " + this.empName );
    }
}
```

Official website: https://www.typescriptlang.org

Typescript docs for beginners : Learn TypeScript using Step-by-Step Tutorials

I recommend once go through this page: Documentation - Do's and Don'ts

Typescript demos: GitHub - ghodelamg/Typescript

Typescript KT Session link : Typescript KT June 2021 - Mukesh - Google Drive

Typescript Videos Drive : Username: typescript.ets@gmail.com Password : ETS@123!