TypeScript Set

TypeScript set is a new data structure added in **ES6** version of JavaScript. It allows us to store **distinct data** (each value occur only once) into the **List** similar to other programming languages. Sets are a bit similar to **maps**, but it stores only **keys**, not the **key-value** pairs.

Create Set

We can create a **set** as below.

```
let mySet = new Set();
```

Set methods

The TypeScript set methods are listed below.

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

Chaining of Set Method

TypeScript set method also allows the chaining of **add()** method. We can understand it from the below example.

Example

```
let studentEntries = new Set();

//Chaining of add() method is allowed in TypeScript
studentEntries.add("John").add("Peter").add("Gayle").add("Kohli");

/Returns Set data
console.log("The List of Set values:");
console.log(studentEntries);
```

Iterating Set Data

We can iterate over set values or entries by using '**for...of**' loop. The following example helps to understand it more clearly.

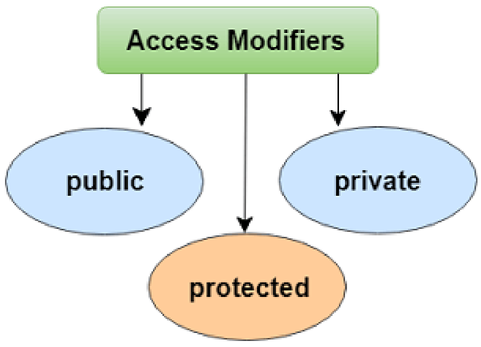
Example

```
let diceEntries = new Set();
diceEntries.add(1).add(2).add(3).add(4).add(5).add(6);

//Iterate over set entries
console.log("Dice Entries are:");
for (let diceNumber of diceEntries) {
   console.log(diceNumber);
}

// Iterate set entries with forEach
console.log("Dice Entries with forEach are:");
diceEntries.forEach(function(value) {
  console.log(value);
});
```

TypeScript Access Modifiers

Like other programming languages, Typescript allows us to use access modifiers at the class level. It gives direct access control to the class member. These class members are functions and properties. We can use class members inside its own class, anywhere outside the class, or within its child or derived class.

The access modifier increases the security of the class members and prevents them from invalid use. We can also use it to control the visibility of data members of a class. If the class does not have to be set any access modifier, TypeScript automatically sets public access modifier to all class members.



Access Modifier	Accessible within class	Accessible in subclass	Accessible externally via class instance
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

TypeScript Accessor

In TypeScript, the accessor property provides a method to access and set the class members. It has two methods which are given below.

1. getter
2. setter

getter

The getter accessor property is the conventional method which is used for retrieving the value of a variable. In object literal, the getter property denoted by "**get**" keyword. It can be public, private, and protected.

Syntax

```
    get propName() {
        // getter, the code executed on getting obj.propName
    },
```

Setter

The setter accessor property is the conventional method which is used for updating the value of a variable. In object literal, the setter property is denoted by "**set**" keyword.

Syntax

```
set propName(value) {
   // setter, the code executed on setting obj.propName = value
 }
```

TypeScript Function

Functions are the fundamental building block of any applications in JavaScript. It makes the code readable, maintainable, and reusable. We can use it to build up layers of abstraction, mimicking classes, information hiding, and modules. In TypeScript, however, we have the concept of classes, namespaces, and modules, but functions still are an integral part in describing how to do things. TypeScript also allows adding new capabilities to the standard JavaScript functions to make the code easier to work.

Advantage of function

These are the main advantages of functions.

- **Code reusability:** We can call a function several times without writing the same block of code again. The code reusability saves time and reduces the program size.
- **Less coding:** Functions makes our program compact. So, we don't need to write many lines of code each time to perform a common task.
- **Easy to debug:** It makes the programmer easy to locate and isolate faulty information.

Function Aspects

There are three aspects of a function.

- **Function declaration:** A function declaration tells the compiler about the function name, function parameters, and return type. The syntax of the function declaration is:

  ```
  function functionName( [arg1, arg2, ...argN] );
  ```

- **Function definition:** It contains the actual statements which are going to executes. It specifies what and how a specific task would be done. The syntax of the function definition is:

  ```
  function functionName( [arg1, arg2, ...argN] ){
      //code to be executed
  }
  ```

- **Function call:** We can call a function from anywhere in the program. The parameter/argument cannot differ in function calling and a function declaration. We must pass the same number of functions as it is declared in the function declaration. The syntax of the function call is:

```
FunctionName();  // function cal
```

Function Creation

We can create a function in two ways. These are:

> Named function:

When we declare and call a function by its given name, then this type of function is known as a **named** function.

Syntax

```
functionName( [arguments] ) { }
```

> Anonymous function:

A function without a name is known as an anonymous function. These type of functions are dynamically declared at runtime. It is defined as an expression. We can store it in a variable, so it does not need function names. Like standard function, it also accepts inputs and returns outputs. We can invoke it by using the variable name, which contains function.

Syntax

```
let res = function( [arguments] ) { }
```

TypeScript Arrow function

ES6 version of TypeScript provides an arrow function which is the **shorthand** syntax for defining the anonymous function, i.e., for function expressions. It omits the function keyword. We can call it fat arrow (because -> is a thin arrow and => is a "**fat**" arrow). It is also called a **Lambda function**. The arrow function has lexical scoping of "**this**" keyword.

The motivation for arrow function is:

- When we don't need to keep typing function.
- It lexically captures the meaning of this keyword.
- It lexically captures the meaning of arguments.

Syntax

We can split the syntax of an Arrow function into three parts:

- **Parameters:** A function may or may not have parameters.

- **The arrow notation/lambda notation** (=>)
- **Statements:** It represents the function's instruction set.

Synatx: (parameter1, parameter2, ..., parameterN) => expression;

Note: If we use the **fat arrow (=>)** notation, there is no need to use the **function** keyword. Parameters are passed in the brackets (), and the function expression is enclosed within the curly brackets {}.

TypeScript Function Overloading

Function overloading is a mechanism or ability to create multiple methods with the **same name** but different parameter types and **return type**. However, it can have the same number of parameters. Function overloading is also known as method overloading.

The Function/Method overloading is allowed when:

- The function name is the same
- The number of parameters is different in each overloaded function.
- The number of parameters is the same, and their type is different.
- The all overloads function must have the same return type.

Advantage of function overloading

- It saves the memory space so that program execution becomes fast.
- It provides code reusability, which saves time and efforts.
- It increases the readability of the program.
- Code maintenance is easy.

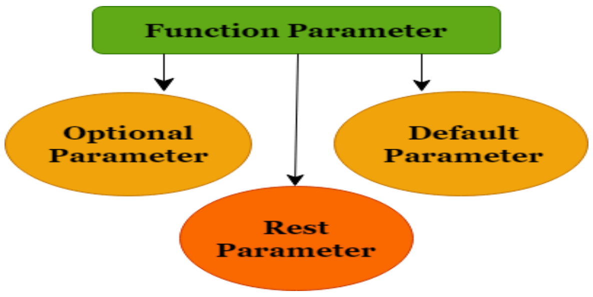
Example

```
//Function with string type parameter
function add(a:string, b:string): string;
//Function with number type parameter
function add(a:number, b:number): number;
//Function Definition
function add(a: any, b:any): any {
   return a + b;
}
//Result
console.log("Addition: " +add("Hello ", "JavaTpoint"));
console.log("Addition: "+add(30, 20));
```

TypeScript Function Parameter

In functions, parameters are the values or arguments that passed to a function. The TypeScript, compiler accepts the same number and type of arguments as defined in the function signature. If the compiler does not match the same parameter as in the function signature, then it will give the compilation error.

We can categorize the function parameter into the three types:



Optional Parameter

JavaScript allows us to call a function without passing any arguments. Hence, in a JavaScript function, the parameter is optional. If we declare the function without passing arguments, then each parameter value is undefined.

Unlike JavaScript, the TypeScript compiler will throw an error if we try to invoke a function without providing the exact number and types of parameters as declared in its function signature. To overcome this problem, TypeScript introduces **optional parameter**.

Syntax

```
function function_name(parameter1[:type], parameter2[:type], parameter3 ? [:type]) { }
```

Example

```
function showDetails(id:number,name:string,e_mail_id?:string) {
  console.log("ID:", id, " Name:",name);
  if(e_mail_id!=undefined)
  console.log("Email-Id:",e_mail_id);
}
showDetails(101,"Virat Kohli");
showDetails(105,"Sachin","sachin@javatpoint.com");
```

Default Parameter

TypeScript provides an option to set default values to the function parameters. If the user does not pass a value to an argument, TypeScript initializes the default value for the parameter. The behavior of the default parameter is the same as an optional parameter. For the default parameter, if a value is not passed in a function call, then the default parameter must follow the required parameters in the function signature. However, if a function signature has a default parameter before a required parameter, we can still call a function which marks the default parameter is passed as an undefined value.

Note: We cannot make the parameter default and optional at the same time.

Syntax

```
function function_name(parameter1[:type], parameter2[:type] = default_value) { }
```

Example

```
function displayName(name: string, greeting: string = "Hello") : string {
   return greeting + ' ' + name + '!';
}
console.log(displayName('JavaTpoint'));   //Returns "Hello JavaTpoint!"
console.log(displayName('JavaTpoint', 'Hi'));   //Returns "Hi JavaTpoint!".
console.log(displayName('Sachin'));    //Returns "Hello Sachin!"
```

Rest Parameter

It is used to pass **zero or more** values to a function. We can declare it by prefixing the **three "dot"** characters ('...') before the parameter. It allows the functions to have a different number of arguments without using the arguments object. The TypeScript compiler will create an array of arguments with the rest parameter so that all array methods can work with the rest parameter. The rest parameter is useful, where we have an undetermined number of parameters.

Rules to follow in rest parameter:

- We can use only one rest parameter in a function.
- It must be an array type.
- It must be the last parameter in a parameter list.

Syntax

```
function function_name(parameter1[:type], parameter2[:type], ...parameter[:type]) { }
```

Example

```
function sum(a: number, ...b: number[]): number {
 let result = a;
 for (var i = 0; i < b.length; i++) {
 result += b[i];
 }
 return result;
}
let result1 = sum(3, 5);
let result2 = sum(3, 5, 7, 9);
console.log(result1 +"\n" + result2);
```