TypeScript Numbers

Like JavaScript, all the numbers in TypeScript are stored as floating-point values. These numeric values are treated like a number data type. The number is used to represents both integers as well as floating-point values. TypeScript also supports Binary(Base 2), Octal(Base 8), Decimal(Base 10), and Hexadecimal(Base 16)literals.

Syntax

```
let identifier: number = value;
```

Number Properties

The Number objects have the following set of properties:

SN	Property_Name	Description
1.	MAX_VALUE	It returns the largest possible value of a number in JavaScript and can have 1.7976931348623157E+308.
2.	MIN_VALUE	It returns the smallest possible value of a number in JavaScript and can have 5E-324.
3.	NEGATIVE_INFINITY	It returns a value that is less than MIN_VALUE.
4.	POSITIVE_INFINITY	It returns a value that is greater than MAX_VALUE.
5.	NaN	When some number calculation is not representable by a valid number, then TypeScript returns a value NaN. It is equal to a value that is not a number.
6.	prototype	It is a static property of the Number object. It is used to assign new properties and methods to the Number object in the current document.

Number Methods

The list of Number methods with their description is given below.

SN	Method_Name	Description
1.	toExponential()	It is used to return the exponential notation in string format.
2.	toFixed()	It is used to return the fixed-point notation in string format.
3.	toLocaleString()	It is used to convert the number into a local specific representation of the number.
4.	toPrecision()	It is used to return the string representation in exponential or fixed-point to the specified precision.
5.	toString()	It is used to return the string representation of the number in the specified base.
6.	valueOf()	It is used to return the primitive value of the number.

Decision Making:

There are various types of Decision making in TypeScript:

→**if statement-** It is a simple form of decision making. It decides whether the statements will be executed or not, i.e., it checks the condition and returns true if the given condition is satisfied.

Syntax:
```
if(condition) {
    // code to be executed
}
```

→**if-else statement-** The if statement only returns the result when the condition is true. But if we want to returns something when the condition is false, then we need to use the if-else statement. The if-else statement tests the condition. If the condition is true, it executes if block and if the condition is false, it executes the else block.

Syntax:

```
if(condition) {
  // code to be executed
} else {
   // code to be executed
}
```

→**if-else-if ladder-** Here a user can take decision among multiple options. It starts execution in a top-down approach. When the condition gets true, it executes the associated statement, and the rest of the condition is bypassed. If it does not find any condition true, it returns the **final else statement.**

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
else{
//code to be executed if all the conditions are false
}
```

→nested if statement- Here, the if statement targets another if statement. The nested if statement means if statement inside the body of another if or else statement.

Syntax:

```
if(condition1) {
    //Nested if else inside the body of "if"
    if(condition2) {
      //Code inside the body of nested "if"
    }
    else {   //Code inside the body of nested "else"   }
}
else {    //Code inside the body of "else."  }
```

TypeScript Switch Statement:

The TypeScript switch statement executes one statement from multiple conditions. It evaluates an expression based on its value that could be Boolean, number, byte, short, int, long, enum type, string, etc. A switch statement has one block of code corresponding to each value. When the match is found, the corresponding block will be executed. A switch statement works like the if-else-if ladder statement.

Syntax

```
switch(expression){

case expression1:
   //code to be executed;
   break;  //optional

case expression2:
   //code to be executed;
   break;  //optional
   ........

default:
   //when no case is matched, this block will be executed;
   break;  //optional
}
```

The switch statement contains the following things. There can be any number of cases inside a switch statement.

Case: The case should be followed by only one constant and then a semicolon. It cannot accept another variable or expression.

Break: The break should be written at the end of the block to come out from the switch statement after executing a case block. If we do not write break, the execution continues with the matching value to the subsequent case block.

Default: The default block should be written at the end of the switch statement. It executes when there are no case will be matched.

TypeScript while loop:

The TypeScript while loop iterates the elements for the infinite number of times. It executes the instruction repeatedly until the specified condition evaluates to true. We can use it when the number of iteration is not known. The while loop syntax is given below.

Syntax

```
while (condition)
{
   //code to be executed
}
```

TypeScript do while loop:

The TypeScript do-while loop iterates the elements for the infinite number of times similar to the while loop. But there is one difference from while loop, i.e., it gets executed at least once whether the condition is true or false. It is recommended to use do-while when the number of iteration is not fixed, and you have to execute the loop at least once. The do-while loop syntax is given below.

Syntax

```
do{
   //code to be executed
}while (condition);
```

TypeScript for loop:

A for loop is a **repetition** control structure. It is used to execute the block of code to a specific number of times. A for statement contains the initialization, condition and increment/decrement in a single line which provides a shorter, and easy to debug structure of looping. The syntax of for loop is given below.

Syntax

```
for (first expression; second expression; third expression ) {
   // statements to be executed repeatedly
}
```

TypeScript for..of loop:

The for..of loop is used to iterate and access the elements of an array, string, set, map, list, or tuple collection. The syntax of the for..of loop is given below.

Syntax

```
for (var val of list) {
  //statements to be executed
}
```

TypeScript for..in loop:

The for..in loop is used with an array, list, or tuple. This loop iterates through a list or collection and returns an index on each iteration. In this, the data type of "**val**" should be a string or any. The syntax of the for..in loop is given below.

Syntax

```
for (var val in list) {
  //statements
}
```

TypeScript Enums:

Enums stands for **Enumerations**. Enums are a new data type supported in TypeScript. It is used to define the set of **named constants**, i.e., a collection of related values. TypeScript supports both **numeric** and **string-based** enums. We can define the enums by using the **enum** keyword.

There are three types of Enums in TypeScript. These are:

- Numeric Enums
- String Enums
- Heterogeneous Enums

TypeScript forEach:T

The forEach() method is an array method which is used to execute a function on ***each item in an array***. We can use it with the JavaScript data types like Arrays, Maps, Sets, etc. It is a useful method for displaying elements in an array.

Syntax

array.forEach(callback[, thisObject]);

→The forEach() method executes the provided **callback** once for each element present in the array in **ascending order**.

TypeScript Map:

TypeScript map is a new data structure added in **ES6** version of JavaScript. It allows us to store data in a **key-value pair** and remembers the original **insertion order** of the keys similar to other programming languages. In TypeScript map, we can use any value either as a **key** or as a **value**.

We can create a map as below.
var map = new Map();

Map methods:

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