Learning Outcome

## Able to develop the real time scenarios based on Node JS applications.

## 

## 

# Typescript Datatype and Operator

## Transcript Datatype

Whenever a variable is created, the intention is to assign some value to that variable but what type of value can be assigned to that variable is dependent upon the datatype of that Variable. In typeScript, type System represents different types of datatypes which are supported by TypeScript. The data type classification is as given below:

Built-in Datatypes: TypeScript has some pre-defined data-types-

|  |  |  |
| --- | --- | --- |
| Built-in Datatype | Keyword | Description |
| Number | number | It is used to represent both Integer as well as Floating-Point numbers |
| Boolean | boolean | Represents true and false |
| String | string | It is used to represent a sequence of characters |
| Void | void | Generally used on function return-types |
| Null | null | It is used when an object does not have any value |

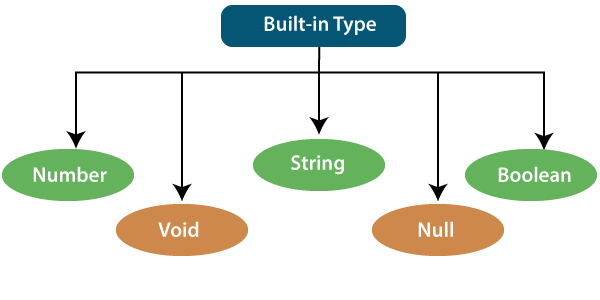


Image : Built-in Datatype

Reference: <https://static.javatpoint.com/tutorial/typescript/images/typescript-types2.png>

### Number

Just like JavaScript, TypeScript supports number data type. All numbers are stored as floating-point numbers. These numbers can be Decimal (base 10), Hexadecimal (base 16) or Octal (base 8).

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

In the above example, let first:number = 1; stores a positive integer as a number. let second: number = 0x37CF; stores a hexadecimal as a number which is equivalent to 14287. When you print this number on your browser's console, it prints the equivalent floating point of the hexadecimal number. let third:number=0377; stores an octal number equivalent to 255.

### Boolean

Boolean values are supported by both JavaScript and TypeScript and stored as true/false values.

TypeScript Boolean:

let isPresent:boolean = true;

Note that, the boolean Boolean is different from the lower case boolean type. The upper-case Boolean is an object type whereas lower case boolean is a primitive type. It is recommended to use the primitive type boolean in your code, because, while JavaScript coerces an object to its primitive type, the TypeScript type system does not. TypeScript treats it like an object type.

So, instead of using upper case function checkExistence(b: Boolean), use the lower case function checkExistence(b: boolean) boolean type.

### String

String is another primitive data type that is used to store text data. String values are surrounded by single quotation marks or double quotation marks.

let employeeName:string = 'John Smith';

//OR

let employeeName:string = "John Smith";

### Void

Similar to languages like Java, void is used where there is no data. For example, if a function does not return any value then you can specify void as return type.

function sayHi(): void {

console.log('Hi!')

}

let speech: void = sayHi();

console.log(speech); //Output: undefined

There is no meaning to assign void to a variable, as only null or undefined is assignable to void.

let nothing: void = undefined;

let num: void = 1; // Error

### Null

TypeScript has a powerful system to deal with null or undefined values. Null and undefined are primitive types and can be used like other types, such as string.

let value: string | undefined | null = null;

console.log(typeof value);

value = 'hello';

console.log(typeof value);

value = undefined;

console.log(typeof value);

Output:

object

string

undefined

## Typescript Operator

An Operator is a symbol which operates on a value or data. It represents a specific action on working with data. The data on which operators operates is called operand. It can be used with one or more than one values to produce a single value. All of the standard JavaScript operators are available with the TypeScript program.



Image : Operators in Typescript

Reference: <https://discoversdkcdn.azureedge.net/postscontent/typeScript/operators%20in%20typescript.png>

Example

10 + 10 = 20;

In the above example, the values '10' and '20' are known as an operand, whereas '+' and '=' are known as operators.

In TypeScript, an operator can be classified into the following ways:

* Arithmetic operators
* Comparison (Relational) operators
* Logical operators
* Bitwise operators
* Assignment operators
* Ternary/conditional operator
* Concatenation operator
* Type Operator

### Arithmetic Operators

Arithmetic operators take numeric values as their operands, performs an action, and then returns a single numeric value. The most common arithmetic operators are addition (+), subtraction (-), multiplication (\*), and division (/).

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Operator\_Name | Description | Example |
| + | Addition | It returns an addition of the values. | let a = 20;  let b = 30;  let c = a + b;  console.log( c ); //  Output  30 |
| - | Subtraction | It returns the difference of the values. | let a = 30;  let b = 20;  let c = a - b;  console.log( c ); //  Output  10 |
| \* | Multiplication | It returns the product of the values. | let a = 30;  let b = 20;  let c = a \* b;  console.log( c ); //  Output  600 |
| / | Division | It performs the division operation, and returns the quotient. | let a = 100;  let b = 20;  let c = a / b;  console.log( c ); //  Output  5 |
| % | Modulus | It performs the division operation and returns the remainder. | let a = 95;  let b = 20;  let c = a % b;  console.log( c ); //  Output  15 |
| ++ | Increment | It is used to increments the value of the variable by one. | let a = 55;  a++;  console.log( a ); //  Output  56 |
| -- | Decrement | It is used to decrements the value of the variable by one. | let a = 55;  a--;  console.log( a ); //  Output  54 |

### Comparison (Relational) Operators

The comparison operators are used to compares the two operands. These operators return a Boolean value true or false. The important comparison operators are given below.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Operator\_Name | Description | Example |
| == | Is equal to | It checks whether the values of the two operands are equal or not. | let a = 10;  let b = 20;  console.log(a==b); //false  console.log(a==10); //true  console.log(10=='10'); //true |
| === | Identical (equal and of the same type) | It checks whether the type and values of the two operands are equal or not. | let a = 10;  let b = 20;  console.log(a===b); //false  console.log(a===10); //true  console.log(10==='10'); //false |
| != | Not equal to | It checks whether the values of the two operands are equal or not. | let a = 10;  let b = 20;  console.log(a!=b); //true  console.log(a!=10); //false  console.log(10!='10'); //false |
| !== | Not identical | It checks whether the type and values of the two operands are equal or not. | let a = 10;  let b = 20;  console.log(a!==b); //true  console.log(a!==10); /false  console.log(10!=='10'); //true |
| > | Greater than | It checks whether the value of the left operands is greater than the value of the right operand or not. | let a = 30;  let b = 20;  console.log(a>b); //true  console.log(a>30); //false  console.log(20> 20'); //false |
| >= | Greater than or equal to | It checks whether the value of the left operands is greater than or equal to the value of the right operand or not. | let a = 20;  let b = 20;  console.log(a>=b); //true  console.log(a>=30); //false  console.log(20>='20'); //true |
| < | Less than | It checks whether the value of the left operands is less than the value of the right operand or not. | let a = 10;  let b = 20;  console.log(a<b); //true  console.log(a<10); //false  console.log(10<'10'); //false |
| <= | Less than or equal to | It checks whether the value of the left operands is less than or equal to the value of the right operand or not. | let a = 10;  let b = 20;  console.log(a<=b); //true  console.log(a<=10); //true  console.log(10<='10'); //true |

### Logical Operators

Logical operators are used for combining two or more condition into a single expression and return the Boolean result true or false. The Logical operators are given below.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Operator\_Name | Description | Example |
| && | Logical AND | It returns true if both the operands(expression) are true, otherwise returns false. | let a = false;  let b = true;  console.log(a&&b); /false  console.log(b&&true); //true  console.log(b&&10); //10 which is also 'true'  console.log(a&&'10'); //false |
| || | Logical OR | It returns true if any of the operands(expression) are true, otherwise returns false. | let a = false;  let b = true;  console.log(a||b); //true  console.log(b||true); //true  console.log(b||10); //true  console.log(a||'10'); //'10' which is also 'true' |
| ! | Logical NOT | It returns the inverse result of an operand(expression). | let a = 20;  let b = 30;  console.log(!true); //false  console.log(!false); //true  console.log(!a); //false  console.log(!b); /false  console.log(!null); //true |

### Bitwise Operators

The bitwise operators perform the bitwise operations on operands. The bitwise operators are as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Operator\_Name | Description | Example |
| & | Bitwise AND | It returns the result of a Boolean AND operation on each bit of its integer arguments. | let a = 2;  let b = 3;  let c = a & b;  console.log(c); //  Output  2 |
| | | Bitwise OR | It returns the result of a Boolean OR operation on each bit of its integer arguments. | let a = 2;  let b = 3;  let c = a | b;  console.log(c); //  Output  3 |
| ^ | Bitwise XOR | It returns the result of a Boolean Exclusive OR operation on each bit of its integer arguments. | let a = 2;  let b = 3;  let c = a ^ b;  console.log(c); //  Output  1 |
| ~ | Bitwise NOT | It inverts each bit in the operands. | let a = 2;  let c = ~ a;  console.log(c); //  Output  -3 |
| >> | Bitwise Right Shift | The left operand's value is moved to the right by the number of bits specified in the right operand. | let a = 2;  let b = 3;  let c = a >> b;  console.log(c); //  Output  0 |
| << | Bitwise Left Shift | The left operand's value is moved to the left by the number of bits specified in the right operand. New bits are filled with zeroes on the right side. | let a = 2;  let b = 3;  let c = a << b;  console.log(c); //  Output  16 |
| >>> | Bitwise Right Shift with Zero | The left operand's value is moved to the right by the number of bits specified in the right operand and zeroes are added on the left side. | let a = 3;  let b = 4;  let c = a >>> b;  console.log(c); //  Output  0 |

### Assignment Operators

Assignment operators are used to assign a value to the variable. The left side of the assignment operator is called a variable, and the right side of the assignment operator is called a value. The data-type of the variable and value must be the same otherwise the compiler will throw an error. The assignment operators are as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Operator\_Name | Description | Example |
| = | Assign | It assigns values from right side to left side operand. | let a = 10;  let b = 5;  console.log("a=b:" +a); //  Output  10 |
| += | Add and assign | It adds the left operand with the right operand and assigns the result to the left side operand. | let a = 10;  let b = 5;  let c = a += b;  console.log(c); //  Output  15 |
| -= | Subtract and assign | It subtracts the right operand from the left operand and assigns the result to the left side operand. | let a = 10;  let b = 5;  let c = a -= b;  console.log(c); //  Output  5 |
| \*= | Multiply and assign | It multiplies the left operand with the right operand and assigns the result to the left side operand. | let a = 10;  let b = 5;  let c = a \*= b;  console.log(c); //  Output  50 |
| /= | Divide and assign | It divides the left operand with the right operand and assigns the result to the left side operand. | let a = 10;  let b = 5;  let c = a /= b;  console.log(c); //  Output  2 |
| %= | Modulus and assign | It divides the left operand with the right operand and assigns the result to the left side operand. | let a = 16;  let b = 5;  let c = a %= b;  console.log(c); //  Output  1 |

### Ternary/Conditional Operator

The conditional operator takes three operands and returns a Boolean value based on the condition, whether it is true or false. Its working is similar to an if-else statement. The conditional operator has right-to-left associativity. The syntax of a conditional operator is given below.

expression ? expression-1 : expression-2;

expression: It refers to the conditional expression.

* expression-1: If the condition is true, expression-1 will be returned.
* expression-2: If the condition is false, expression-2 will be returned.

Example

let num = 16;

let result = (num > 0) ? "True":"False"

console.log(result);

Output:

True

### Concatenation Operator

The concatenation (+) operator is an operator which is used to append the two string. In concatenation operation, we cannot add a space between the strings. We can concatenate multiple strings in a single statement. The following example helps us to understand the concatenation operator in TypeScript.

Example

let message = "Welcome to " + "EduNet Foundation";

console.log("Result of String Operator: " +message);

Output:

Result of String Operator: Welcome to EduNet Foundation

### Type Operators

There are a collection of operators available which can assist you when working with objects in TypeScript. Operators such as typeof, instanceof, in, and delete are the examples of Type operator. The detail explanation of these operators is given below.

|  |  |  |
| --- | --- | --- |
| Operator\_Name | Description | Example |
| In | It is used to check for the existence of a property on an object. | let Bike = {make: 'Honda', model: 'CLIQ', year: 2018};  console.log('make' in Bike); //  Output:  True |
| Delete | It is used to delete the properties from the objects. | let Bike = { Company1: 'Honda',  Company2: 'Hero',  Company3: 'Royal Enfield'  };  delete Bike.Company1;  console.log(Bike); //  Output:  { Company2: 'Hero', Company3: 'Royal Enfield' } |
| Typeof | It returns the data type of the operand. | let message = "Welcome to " + "JavaTpoint";  console.log(typeof message); //  Output:  String |
| Instanceof | It is used to check if the object is of a specified type or not. | let arr = [1, 2, 3];  console.log( arr instanceof Array ); // true  console.log( arr instanceof String ); // false |

# Type Script String & Tuple, Oops

## Typescript String

In TypeScript, the string is an object which represents the sequence of character values. It is a primitive data type which is used to store text data. The string values are surrounded by single quotation mark or double quotation mark. An array of characters works the same as a string.

Syntax

let var\_name = new String(string);

Example

let uname = new String("Hello Edunet");

console.log("Message: " +uname);

console.log("Length: "+uname.length);

Output:

Message: Hello Edunet

Length: 11

There are three ways in which we can create a string.

#### Single quoted strings

It enclosed the string in a single quotation mark, which is given below.

 Example

var studentName: String = 'Peter';

#### Double quoted strings

It enclosed the string in double quotation marks, which is given below.

Example:

var studentName: String = "Peter";

#### Back-ticks strings

It is used to write an expression. We can use it to embed the expressions inside the string. It is also known as Template string. TypeScript supports Template string from ES6 version.

Example:

let empName:string = "Rohit Sharma";

let compName:string = "Edunet";

// Pre-ES6

let empDetail1: string = empName + " works in the " + compName + " company.";

// Post-ES6

let empDetail2: string = `${empName} works in the ${compName} company.`;

console.log("Before ES6: " +empDetail1);

console.log("After ES6: " +empDetail2);

Output:

Before ES6: Rohit Sharma works in the Edunet company.

After ES6: Rohit Sharma works in the Edunet company.

#### Multi-Line String

ES6 provides us to write the multi-line string. We can understand it from the below example.

Example

let multi = 'hello ' +

'world ' +

'my ' +

'name ' +

'is ' +

'Rohit';

If we want that each line in the string contains "new line" characters, then we have to add "\n" at the end of each string.

Example

let multi = ' hello\n ' +

'Students\n ' +

'my\n ' +

'name\n ' +

'is\n ' +

'Rohit Sharma';

console.log(multi);

Output:

hello

Students

my

name

is

Rohit Sharma

#### String Literal Type

A string literal is a sequence of characters enclosed in double quotation marks (" "). It is used to represent a sequence of character which forms a null-terminated string. It allows us to specify the exact string value specified in the "string literal type." It uses "pipe" or " | " symbol between different string value.

Syntax

Type variableName = "value1" | "value2" | "value3"; // upto N number of values

#### String Methods

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

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | charAt() | It returns the character of the given index. |
|  | concat() | It returns the combined result of two or more string. |
|  | endsWith() | It is used to check whether a string ends with another string. |
|  | includes() | It checks whether the string contains another string or not. |
|  | indexOf() | It returns the index of the first occurrence of the specified substring from a string, otherwise returns -1. |
|  | lastIndexOf() | It returns the index of the last occurrence of a value in the string. |
|  | match() | It is used to match a regular expression against the given string. |
|  | replace() | It replaces the matched substring with the new substring. |
|  | search() | It searches for a match between a regular expression and string. |
|  | slice() | It returns a section of a string. |
|  | split() | It splits the string into substrings and returns an array. |
|  | substring() | It returns a string between the two given indexes. |
|  | toLowerCase() | It converts the all characters of a string into lower case. |
|  | toUpperCase() | It converts the all characters of a string into upper case. |
|  | trim() | It is used to trims the white space from the beginning and end of the string. |
|  | trimLeft() | It is used to trims the white space from the left side of the string. |
|  | trimRight() | It is used to trims the white space from the right side of the string. |
|  | valueOf() | It returns a primitive value of the specified object. |

Example

//String Initialization

let str1: string = 'Hello';

let str2: string = 'Edunet';

//String Concatenation

console.log("Combined Result: " +str1.concat(str2));

//String charAt

console.log("Character At 4: " +str2.charAt(4));

//String indexOf

console.log("Index of T: " +str2.indexOf('T'));

//String replace

console.log("After Replacement: " +str1.replace('Hello', 'Welcome to'));

//String uppercase

console.log("UpperCase: " +str2.toUpperCase());

Output:

Combined Result: HelloJavaTpoint

Character At 4: T

Index of T: 4

After Replacement: Welcome to

UpperCase: EDUNET

## Typescript Tuple

We know that an array holds multiple values of the same data type. But sometimes, we may need to store a collection of values of different data types in a single variable. Arrays will not provide this feature, but TypeScript has a data type called Tuple to achieve this purpose. A Tuple is an array which store multiple fields belong to different data types. It is similar to the structures in the C programming language.

A tuple is a data type which can be used like any other variables. It represents the heterogeneous collection of values and can also be passed as parameters in a function call.

In abstract mathematics, the term tuple is used to denote a multi-dimensional coordinate system. JavaScript does not have tuple as data type, but tuples are available in TypeScript. The order of elements in a tuple is important.

Syntax

let tuple\_name = [val1,val2,val3, ...val n];

Example

let arrTuple = [101, "Edunet", 105, "Abhishek"];

console.log(arrTuple);

Output:

[101, 'Edunet', 105, 'Abhishek']

We can also declare and initialize a tuple separately by initially declaring the tuple as an empty tuple in Typescript.

Example

let arrTuple = [];

arrTuple[0] = 101

arrTuple[1] = 105

### Accessing tuple Elements

We can read or access the fields of a tuple by using the index, which is the same as an array. In Tuple, the index starts from zero.

Example:

let empTuple = ["Rohit Sharma", 25, "Edunet"];

console.log("Name of the Employee is : "+empTuple [0]);

console.log("Age of the Employee is : "+empTuple [1]);

console.log(empTuple [0]+" is working in "+empTuple [2]);

Output:

Name of the Employee is: Rohit Sharma

Age of the Employee is: 25

Rohit Sharma is working in Edunet

### Operations on Tuple

A tuple has two operations:

* Push()
* Pop()

**Push()**

The push operation is used to add an element to the tuple.

Example:

let empTuple = ["Rohit Sharma", 25, "Edunet"];

console.log("Items: "+empTuple);

console.log("Length of Tuple Items before push: "+empTuple.length); // returns the tuple size

empTuple.push(10001); // append value to the tuple

console.log("Length of Tuple Items after push: "+empTuple.length);

console.log("Items: "+empTuple);

Output:

Items: Rohit Sharma, 25, Edunet

Length of Tuple Items before push: 3

Length of Tuple Items after push: 4

Items: Rohit Sharma, 25, Edunet, 10001

Pop()

The pop operation is used to remove an element from the tuple.

Example

let empTuple = ["Rohit Sharma", 25, "Edunet", 10001];

console.log("Items: "+empTuple);

console.log("Length of Tuple Items before pop: "+empTuple.length); // returns the tuple size

empTuple.pop(); // removed value to the tuple

console.log("Length of Tuple Items after pop: "+empTuple.length);

console.log("Items: "+empTuple);

Output:

Items: Rohit Sharma,25, Edunet, 10001

Length of Tuple Items before pop: 4

Length of Tuple Items after pop: 3

Items: Rohit Sharma, 25, Edunet

### Update or Modify the Tuple Elements

Tuples are mutable, which means we can update or change the values of tuple elements. To modify the fields of a Tuple, we need to use the index of the fields and assignment operator. We can understand it with the following example.

Example:

let empTuple = ["Rohit Sharma", 25, "Edunet"];

empTuple[1] = 30;

console.log("Name of the Employee is: "+empTuple [0]);

console.log("Age of the Employee is: "+empTuple [1]);

console.log(empTuple [0]+" is working in "+empTuple [2]);

Output:

Name of the Employee is: Rohit Sharma

Age of the Employee is: 30

Rohit Sharma is working in Edunet

### Clear the fields of a Tuple

We cannot delete the tuple variable, but its fields could be cleared. To clear the fields of a tuple, assign it with an empty set of tuple field, which is shown in the following example.

Example:

let empTuple = ["Rohit Sharma", 25, "Edunet"];

empTuple = [];

console.log(empTuple);

Output:

[]

### Destructuring the Tuple

Destructuring allows us to break up the structure of an entity. TypeScript used destructuring in the context of a tuple.

Example:

let empTuple = ["Rohit Sharma", 25, "Edunet"];

let [emp, student] = empTuple;

console.log(emp);

console.log(student);

Output:

Rohit Sharma

25

### Passing Tuple to Functions

We can pass a tuple to functions, which can be shown in the below example.

Example:

//Tuple Declaration

let empTuple = ["EduNet", 101, "Abhishek"];

//Passing tuples in function

function display(tuple\_values:any[]) {

for(let i = 0;i<empTuple.length;i++) {

console.log(empTuple[i]);

}

}

//Calling tuple in function

display(empTuple);

Output:

EduNet

101

Abhishek

## Typescript OOPS

Object Oriented Programming or OOP is a programming paradigm that has four principles which are:

* Inheritance,
* Abstraction,
* Polymorphism,
* And Encapsulation.

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). … In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

At the heart of OOP is the concept of an object which may refer to an abstract or concrete object in our world so this makes it easier for programmers to model actual problems with computers languages before trying to find the solutions.

An object can have data (which form the properties or attributes of the real-world object) and code in a form of methods (which represent the behavior in the equivalent real-world object). Think of a car for example, it has a color, weight and speed and can move forward or backward.

You can very easily create a “Car” object with an OOP language such as TypeScript to represent this car in the computer memory.

Class-based programming, or more commonly class-orientation, is a style of Object-oriented programming(OOP) in which inheritance occurs via defining classes of objects, instead of inheritance occurring via the objects alone (compare prototype-based programming).

Unlike JavaScript which has a prototype-based OOP, TypeScript is a class-based OOP language.

In a programming language, a class has the same meaning in the sense that it represents a category of objects or a type but it also has a concrete form as an extensible template of code for creating objects via instantiation.

et’s refer back to our “Car” object. Before, you can create the object, you need to create the “Car” class which contains the data fields and methods (procedures which are attached to the class) that each car has, next, you instantiate the class to create one or more objects (cars).

We refer to Inheritance in OOP when a class A inherits the properties of another class B. A is also said to extend B. This is a familiar behavior in nature where the children of humans or other creatures inherit the traits of their parents. \*\* Thanks to inheritance, we can reuse the fields and methods of the existing class which facilitates reusability, one of the goals of the OOP paradigm.

Inheritance is a relationship between two classes where the class that is used as the basis for inheritance is refereed to as a \*superclass or base class. While the class that inherits from a base class is refereed to as a subclass.

In TypeScript, we use the extends keyword for defining an inheritance.

Inheritance implies Polymorphism (Another fundamental principle of OOP). Think of that, when a class B and C inherit the methods of a class A. They can customize or change the inherited methods as necessary. For example, both a Plane and a Car inherit a move() method from a Vehicle but the move behavior of a Plane is actually the flying instead of moving using wheels. So when we create the Plane class that extends the Vehicle class, we need to override the move() method to implement a flying behavior instead of the regular movement.

Overriding the inherited (parent) method and re-implementing its behavior is what refers to Polymorphism. In fact, the meaning of polymorphism from the Greek origin is when something occurs in many different forms.

What about Encapsulation?

Encapsulation represents another principle of Object-oriented programming. The concept refers to the grouping of data variables and methods. The class in OOP languages \*\*\*\*enables encapsulation via providing the way to group data and methods.

Encapsulation is also used to hide data and methods that are meant to be internal and only required for the inner working of the object. In this meaning, encapsulation is equivalent to Abstraction, another fundamental principle of OOP.

TypeScript provides the programmer with access modifiers or keywords like public, protected and private to specify the degree of visibility of the class members to the outside.

# Object and Classes

TypeScript is object-oriented JavaScript. TypeScript supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Typescript gives built in support for this concept called class. JavaScript ES5 or earlier didn’t support classes. Typescript gets this feature from ES6.

## Creating classes

Use the class keyword to declare a class in TypeScript. The syntax for the same is given below −

Syntax

class class\_name {

//class scope

}

The class keyword is followed by the class name. The rules for identifiers must be considered while naming a class.

A class definition can include the following −

* Fields − A field is any variable declared in a class. Fields represent data pertaining to objects
* Constructors − Responsible for allocating memory for the objects of the class
* Functions − Functions represent actions an object can take. They are also at times referred to as methods

These components put together are termed as the data members of the class.

Consider a class Person in typescript.

class Person {

}

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Person = (function () {

function Person() {

}

return Person;

}());

Example: Declaring a class

class Car {

//field

engine:string;

//constructor

constructor(engine:string) {

this.engine = engine

}

//function

disp():void {

console.log("Engine is : "+this.engine)

}

}

The example declares a class Car. The class has a field named engine. The var keyword is not used while declaring a field. The example above declares a constructor for the class.

A constructor is a special function of the class that is responsible for initializing the variables of the class. TypeScript defines a constructor using the constructor keyword. A constructor is a function and hence can be parameterized.

The this keyword refers to the current instance of the class. Here, the parameter name and the name of the class’s field are the same. Hence to avoid ambiguity, the class’s field is prefixed with the this keyword.

disp() is a simple function definition. Note that the function keyword is not used here.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Car = (function () {

//constructor

function Car(engine) {

this.engine = engine;

}

//function

Car.prototype.disp = function () {

console.log("Engine is : " + this.engine);

};

return Car;

}());

### Creating Instance objects

To create an instance of the class, use the new keyword followed by the class name. The syntax for the same is given below −

Syntax

var object\_name = new class\_name([ arguments ])

The new keyword is responsible for instantiation.

The right-hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized.

Example: Instantiating a class

var obj = new Car("Engine 1")

### Accessing Attributes and Functions

A class’s attributes and functions can be accessed through the object. Use the ‘ . ’ dot notation (called as the period) to access the data members of a class.

//accessing an attribute

obj.field\_name

//accessing a function

obj.function\_name()

Example: Putting them together

class Car {

//field

engine:string;

//constructor

constructor(engine:string) {

this.engine = engine

}

//function

disp():void {

console.log("Function displays Engine is : "+this.engine)

}

}

//create an object

var obj = new Car("XXSY1")

//access the field

console.log("Reading attribute value Engine as : "+obj.engine)

//access the function

obj.disp()

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Car = (function () {

//constructor

function Car(engine) {

this.engine = engine;

}

//function

Car.prototype.disp = function () {

console.log("Function displays Engine is : " + this.engine);

};

return Car;

}());

//create an object

var obj = new Car("XXSY1");

//access the field

console.log("Reading attribute value Engine as : " + obj.engine);

//access the function

obj.disp();

The output of the above code is as follows −

Reading attribute value Engine as : XXSY1

Function displays Engine is : XXSY1

# Inheritance & Interface

An interface can be extended by other interfaces. In other words, an interface can inherit from other interface. Typescript allows an interface to inherit from multiple interfaces.



Image : Inheritance & Interface

Reference: <https://www.tutorialspoint.com/typescript/images/interface_and_objects.jpg>

Use the extends keyword to implement inheritance among interfaces.

## Single Interface Inheritance

Syntax:

Child\_interface\_name extends super\_interface\_name

## Multiple Interface Inheritance

Syntax:

Child\_interface\_name extends super\_interface1\_name,

super\_interface2\_name,…,super\_interfaceN\_name

Example: Simple Interface Inheritance

interface Person {

age:number

}

interface Musician extends Person {

instrument:string

}

var drummer = <Musician>{};

drummer.age = 27

drummer.instrument = "Drums"

console.log("Age: "+drummer.age) console.log("Instrument: "+drummer.instrument)

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var drummer = {};

drummer.age = 27;

drummer.instrument = "Drums";

console.log("Age: " + drummer.age);

console.log("Instrument: " + drummer.instrument);

Its output is as follows −

Age: 27

Instrument: Drums

Example: Multiple Interface Inheritance

interface IParent1 {

v1:number

}

interface IParent2 {

v2:number

}

interface Child extends IParent1, IParent2 { }

var Iobj:Child = { v1:12, v2:23}

console.log("value 1: "+this.v1+" value 2: "+this.v2)

The object Iobj is of the type interface leaf. The interface leaf by the virtue of inheritance now has two attributes- v1 and v2 respectively. Hence, the object Iobj must now contain these attributes.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Iobj = { v1: 12, v2: 23 };

console.log("value 1: " + this.v1 + " value 2: " + this.v2);

The output of the above code is as follows −

value 1: 12 value 2: 23

# Angular js Child Component

In AngularJS, a Component is a special kind of directive that uses a simpler configuration which is suitable for a component-based application structure.

This makes it easier to write an app in a way that's similar to using Web Components or using the new Angular's style of application architecture.

## Advantages of Components:

* simpler configuration than plain directives
* promote sane defaults and best practices
* optimized for component-based architecture
* writing component directives will make it easier to upgrade to Angular

## When not to use Components:

* for directives that need to perform actions in compile and pre-link functions, because they aren't available
* when you need advanced directive definition options like priority, terminal, multi-element
* when you want a directive that is triggered by an attribute or CSS class, rather than an element

## Creating and configuring a Component

Components can be registered using the .component() method of an AngularJS module (returned by angular.module()). The method takes two arguments:

* The name of the Component (as string).
* The Component config object. (Note that, unlike the .directive() method, this method does not take a factory function.)

## Component-based application architecture

As already mentioned, the component helper makes it easier to structure your application with a component-based architecture. But what makes a component beyond the options that the component helper has?

* Components only control their own View and Data: Components should never modify any data or DOM that is out of their own scope. Normally, in AngularJS it is possible to modify data anywhere in the application through scope inheritance and watches. This is practical, but can also lead to problems when it is not clear which part of the application is responsible for modifying the data. That is why component directives use an isolate scope, so a whole class of scope manipulation is not possible.
* Components have a well-defined public API - Inputs and Outputs: However, scope isolation only goes so far, because AngularJS uses two-way binding. So if you pass an object to a component like this - bindings: {item: '='}, and modify one of its properties, the change will be reflected in the parent component. For components however, only the component that owns the data should modify it, to make it easy to reason about what data is changed, and when. For that reason, components should follow a few simple conventions

Example of a component tree

The following example expands on the simple component example and incorporates the concepts we introduced above:

Instead of an ngController, we now have a heroList component that holds the data of different heroes, and creates a heroDetail for each of them.

The heroDetail component now contains new functionality:

* a delete button that calls the bound onDelete function of the heroList component
* an input to change the hero location, in the form of a reusable editableField component. Instead of manipulating the hero object itself, it sends a changeset upwards to the heroDetail, which sends it upwards to the heroList component, which updates the original data.

Components as route templates

Components are also useful as route templates (e.g. when using ngRoute). In a component-based application, every view is a component:

var myMod = angular.module('myMod', ['ngRoute']);

myMod.component('home', {

template: '<h1>Home</h1><p>Hello, {{ $ctrl.user.name }} !</p>',

controller: function() {

this.user = {name: 'world'};

}

});

myMod.config(function($routeProvider) {

$routeProvider.when('/', {

template: '<home></home>'

});

});

When using $routeProvider, you can often avoid some boilerplate, by passing the resolved route dependencies directly to the component. Since 1.5, ngRoute automatically assigns the resolves to the route scope property $resolve (you can also configure the property name via resolveAs). When using components, you can take advantage of this and pass resolves directly into your component without creating an extra route controller:

var myMod = angular.module('myMod', ['ngRoute']);

myMod.component('home', {

template: '<h1>Home</h1><p>Hello, {{ $ctrl.user.name }} !</p>',

bindings: {

user: '<'

}

});

myMod.config(function($routeProvider) {

$routeProvider.when('/', {

template: '<home user="$resolve.user"></home>',

resolve: {

user: function($http) { return $http.get('...'); }

}

});

});

# Angular js Data Binding, Event Binding,2 Way Binding

## Angular js Data Binding

Data-binding in AngularJS apps is the automatic synchronization of data between the model and view components. The way that AngularJS implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa.



Image : Angular js Data Binding

Reference: <https://docs.angularjs.org/img/One_Way_Data_Binding.png>

Most templating systems bind data in only one direction: they merge template and model components together into a view. After the merge occurs, changes to the model or related sections of the view are NOT automatically reflected in the view. Worse, any changes that the user makes to the view are not reflected in the model. This means that the developer has to write code that constantly syncs the view with the model and the model with the view.

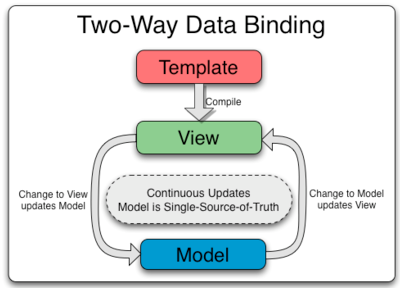


Image : Angular js 2-way Data Binding

Reference: <https://www.javatpoint.com/js/angularjs/images/two-way-data-binding.png>

AngularJS templates work differently. First the template (which is the uncompiled HTML along with any additional markup or directives) is compiled on the browser. The compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. The model is the single-source-of-truth for the application state, greatly simplifying the programming model for the developer. You can think of the view as simply an instant projection of your model.

Because the view is just a projection of the model, the controller is completely separated from the view and unaware of it. This makes testing a snap because it is easy to test your controller in isolation without the view and the related DOM/browser dependency.

## How To Bind Events In AngularJS

We want the user to be able to take an action, and cause something to happen on the page. Users will enter text into input boxes, pick items from lists and click buttons. These types of user actions result in a flow of data from an element to a component. Listening for certain events such as keystrokes, mouse movements, and clicks are done with Angular event binding.

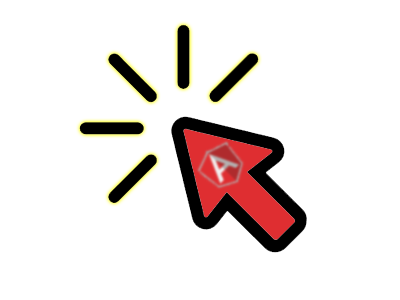


Image : How To Bind Events In AngularJS

Reference: <https://vegibit.com/wp-content/uploads/2018/11/How-To-Bind-Events-In-AngularJS.png>

The most common action a user may take is a click event. Some of the common JavaScript events include onclick, onmouseover, onmouseout, onchange, onkeydown, onkeyup, and many more. We can set up a click event in Angular using a special syntax.

### (click)=”methodToRun()”

In the virtual-machines.component.html Angular Template, if you would like to respond to a click event you can use this form of binding.

<button

[disabled]="!allowNewVm"

class="btn btn-lg"

(click)="onCreateVM()"

>Add a VM</button>

The (click) says, “hey, we’re listening for the user to click”. The =”onCreateVM()” portion of the markup says, “run this piece of code when clicked”.

#### Configure The .ts file to respond

In our template, we now have a click event listener set up. When it is click, we want something to happen. To set this up, first we will create a new variable to hold an initial state of data. This will be the vmCreated variable we see here in virtual-machines.component.ts.

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-virtual-machines',

templateUrl: './virtual-machines.component.html',

styleUrls: ['./virtual-machines.component.css'],

})

export class VirtualMachinesComponent implements OnInit {

allowNewVm = false;

vmCreated = 'Initial State: Add a VM?';

constructor() {

setTimeout(() => {

this.allowNewVm = true

}, 1500);

}

ngOnInit() {

}

}

In addition to the new variable, we also need the new method or function that is supposed to run when the button is clicked. We had named that function onCreateVM in the template. That means we will add that named function to the TypeScript file like so:

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-virtual-machines',

templateUrl: './virtual-machines.component.html',

styleUrls: ['./virtual-machines.component.css'],

})

export class VirtualMachinesComponent implements OnInit {

allowNewVm = false;

vmCreated = 'Initial State: Add a VM?';

constructor() {

setTimeout(() => {

this.allowNewVm = true

}, 1500);

}

ngOnInit() {

}

onCreateVM() {

this.vmCreated = 'Button Clicked: New VM spun up!';

}

}

Reference The Data in The Template

Finally, we output the contents of the vmCreated variable via string interpolation. On page load, it should output the initial value of the variable. When the user clicks the button, that value should change which will also update the user interface in real-time.

<button [disabled]="!allowNewVm" class="btn btn-lg" (click)="onCreateVM()">Add a VM</button>

the allowNewVm variable is currently <b [innerText]="allowNewVm"></b>

<hr>

<h5>{{ vmCreated }}</h5>

<app-virtual-machine></app-virtual-machine>

Reference The Data in The Template

Finally, we output the contents of the vmCreated variable via string interpolation. On page load, it should output the initial value of the variable. When the user clicks the button, that value should change which will also update the user interface in real-time.

<button [disabled]="!allowNewVm" class="btn btn-lg" (click)="onCreateVM()">Add a VM</button>

the allowNewVm variable is currently <b [innerText]="allowNewVm"></b>

<hr>

<h5>{{ vmCreated }}</h5>

<app-virtual-machine></app-virtual-machine>

### (input)=”onMethodToRun($event)”

Speaking of user input, let’s set up an <input> field that will allow the user to provide a name for the VM before creating it. That $event is special. It is like a reserved variable name you can use in the template during event binding. It is a way to fetch the data from that input when it fires.

## Two Way Data Binding

In the code above, we kind of took the long route to set up data binding. An easier approach is to simply use the [(ngModel)] directive.

<form>

<div class="form-group">

<label for="vmname">VM Name</label>

<input

type="text"

class="form-control"

[(ngModel)]="vmName"

name="vmName"

id="vmname"

>

<small class="form-text text-muted">Enter the name for a new VM.</small>

</div>

</form>

<p>Add the <b>{{ vmName }}</b> virtual machine?</p>

<button [disabled]="!allowNewVm" class="btn btn-lg" (click)="onCreateVM()">Add a VM</button>

the allowNewVm variable is currently <b [innerText]="allowNewVm"></b>

<hr>

<h5>{{ vmCreated }}</h5>

<app-virtual-machine></app-virtual-machine> You’ll note that we also set the name attribute using name=”vmName”. The reason for this is because if you leave it out, you may see an error like “ERROR Error: If ngModel is used within a form tag, either the name attribute must be set or the formcontrol must be defined as ‘standalone’ in ngModelOptions.”

Testing out the data binding using [(ngModel)]=”vmName” does appear to be working well. Also note that we no longer need the onSetVmName(event) method in virtual-machines.component.ts since ngModel is handling that for us automatically.

References

1. <https://www.tutorialsteacher.com/typescript/for-loop>
2. <https://docs.angularjs.org/guide/component>
3. <https://angular.io/guide/event-binding>
4. <https://vegibit.com/how-to-bind-events-in-angularjs/>
5. <https://www.javatpoint.com/angularjs-data-binding>
6. <https://www.tutorialspoint.com/typescript/typescript_interfaces.htm>
7. <https://www.techiediaries.com/object-oriented-programming-concepts/>
8. <https://www.w3schools.blog/inheritance-typescript>
9. <https://docs.angularjs.org/guide/databinding>
10. <https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html>