TYPESCRIPT

## What is TypeScript?

* By definition, “TypeScript is JavaScript for application-scale development.”
* TypeScript is a strongly typed, object oriented, compiled language.
* TypeScript is both a language and a set of tools.
* TypeScript is a typed superset of JavaScript compiled to JavaScript. In other words, TypeScript is JavaScript plus some additional features.

## Why TypeScript?

* TypeScript is just JavaScript.
* Javascript is typescript.
* Typescript is portable
* Compilation
* Type checking
* supports Object Oriented Programming.
* Helps java and c# developers to adapt typescript.

## Local Environment Setup

* The source files are typically named with the extension **.ts**
* The TSC (TypeScript Compiler) is a source-to-source compiler (transcompiler / transpiler).
* The TSC generates a JavaScript version of the **.ts** file passed to it.
* The compiler rejects any raw JavaScript file passed to it. The compiler deals with only **.ts**
* **Step 1** − Download and install NodeJS.
* **Step 2** − To verify if the installation was successful, enter the command ***node –v*** in the terminal window.
* **Step 3** − Type the following command in the terminal window to install TypeScript.
* npm install -g typescript
* **Step 4** − Try opening a terminal and runthe below command to see if it has been properly installed.
* tsc –v
* **Step 5** − compile the file using the following command on the terminal window.
* tsc Test.ts
* **Step 6** − The file is compiled to Test.js. To run the program written, type the following in the terminal.
* Node test.js

NP:Multiple files can be compiled at once.

* tsc file1.ts, file2.ts, file3.ts

Compiler Flags

Compiler flags enable you to change the behavior of the compiler during compilation. Each compiler flag exposes a setting that allows you to change how the compiler behaves.

The following table lists some common flags associated with the TSC compiler. A typical command-line usage uses some or all switches.

|  |  |
| --- | --- |
| **S.No.** | **Compiler flag & Description** |
| 1. | **--help**  Displays the help manual |
| 2. | **--module**  Load external modules |
| 3. | **--target**  Set the target ECMA version |
| 4. | **--declaration**  Generates an additional .d.ts file |
| 5. | **--removeComments**  Removes all comments from the output file |
| 6. | **--out**  Compile multiple files into a single output file |
| 7. | **--sourcemap**  Generate a sourcemap (.map) files |
| 8. | **--module noImplicitAny**  Disallows the compiler from inferring the any type |
| 9. | **--watch**  Watch for file changes and recompile them on the fly |

Identifiers in TypeScript

Identifiers are names given to elements in a program like variables, functions etc. The rules for identifiers are −

* Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.
* Identifiers cannot include special symbols except for underscore (\_) or a dollar sign ($).
* Identifiers cannot be keywords.
* They must be unique.
* Identifiers are case-sensitive.
* Identifiers cannot contain spaces.

The following tables lists a few examples of valid and invalid identifiers −

|  |  |
| --- | --- |
| **Valid identifiers** | **Invalid identifiers** |
| firstName | Var |
| first\_name | first name |
| num1 | first-name |
| $result | 1number |

TypeScript ─ Keywords

Keywords have a special meaning in the context of a language. The following table lists some keywords in TypeScript.

|  |  |  |  |
| --- | --- | --- | --- |
| break | as | any | switch |
| case | if | throw | else |
| var | number | string | get |
| module | type | instanceof | typeof |
| public | private | enum | export |
| finally | for | while | void |
| null | super | this | new |
| in | return | true | false |
| any | extends | static | let |
| package | implements | interface | function |
| new | try | yield | const |
| continue | do | catch |  |

Whitespace and Line Breaks

TypeScript ignores spaces, tabs, and newlines that appear in programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

TypeScript is Case-sensitive

TypeScript is case-sensitive. This means that TypeScript differentiates between uppercase and lowercase characters.

Semicolons are optional

Each line of instruction is called a **statement**. Semicolons are optional in TypeScript.

**Example**

console.log("hello world")

console.log("We are learning TypeScript")

A single line can contain multiple statements. However, these statements must be separated by a semicolon.

Comments in TypeScript

Comments are a way to improve the readability of a program. Comments can be used to include additional information about a program like author of the code, hints about a function/ construct etc. Comments are ignored by the compiler.

TypeScript supports the following types of comments −

* **Single-line comments ( // )** − Any text between a // and the end of a line is treated as a comment
* **Multi-line comments (/\* \*/)** − These comments may span multiple lines.

**Example**

//this is single line comment

/\* This is a

Multi-line comment

\*/

**DataTypes:**

|  |  |  |
| --- | --- | --- |
| **Data type** | **Keyword** | **Description** |
| Number | number | Double precision 64-bit floating point values. It can be used to represent both, integers and fractions. |
| String | string | Represents a sequence of Unicode characters |
| Boolean | boolean | Represents logical values, true and false |
| Void | void | Used on function return types to represent non-returning functions |
| Null | null | Represents an intentional absence of an object value. |
| Undefined | undefined | Denotes value given to all uninitialized variables |
| Any | any | The any data type is the super type of all types in TypeScript. It denotes a dynamic type |

### **User-defined Types**

User-defined types include the following things.

* Enumerations (enums)
* classes
* interfaces
* arrays
* tuple

TypeScript - Variables

A variable, by definition, is “a named space in the memory” that stores values. In other words, it acts as a container for values in a program. TypeScript variables must follow the JavaScript naming rules −

* Variable names can contain alphabets and numeric digits.
* They cannot contain spaces and special characters, except the underscore (\_) and the dollar ($) sign.
* Variable names cannot begin with a digit.

A variable must be declared before it is used. Use the **var** keyword to declare variables.

Variable Declaration in TypeScript

The type syntax for declaring a variable in TypeScript is to include a colon (:) after the variable name, followed by its type. Just as in JavaScript, we use the **var** keyword to declare a variable.

When you declare a variable, you have four options −

* Declare its type and value in one statement.
* Declare its type but no value. In this case, the variable will be set to undefined.
* Declare its value but no type. The variable type will be set to any.
* Declare neither value not type. In this case, the data type of the variable will be any and will be initialized to undefined.

The following table illustrates the valid syntax for variable declaration as discussed above −

|  |  |
| --- | --- |
| **S.No.** | **Variable Declaration Syntax & Description** |
| 1. | **var name:string = ”mary”**  The variable stores a value of type string |
| 2. | **var name:string;**  The variable is a string variable. The variable’s value is set to undefined by default |
| 3. | **var name = ”mary”**  The variable’s type is inferred from the data type of the value. Here, the variable is of the type string |
| 4. | **var name;**  The variable’s data type is any. Its value is set to undefined by default. |

Example: Variables in TypeScript

varname:string="John";

var score1:number=50;

var score2:number=42.50

var sum = score1 + score2

console.log("name"+name)

console.log("first score: "+score1)

console.log("second score: "+score2)

console.log("sum of the scores: "+sum)

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var name ="John";

var score1 =50;

var score2 =42.50;

var sum = score1 + score2;

console.log("name"+ name);

console.log("first score: "+ score1);

console.log("second score : "+ score2);

console.log("sum of the scores: "+ sum);

The output of the above program is given below −

name:John

first score:50

second score:42.50

sum of the scores:92.50

The TypeScript compiler will generate errors, if we attempt to assign a value to a variable that is not of the same type. Hence, TypeScript follows Strong Typing. The Strong typing syntax ensures that the types specified on either side of the assignment operator (=) are the same. This is why the following code will result in a compilation error −

varnum:number="hello"// will result in a compilation error

TypeScript Variable Scope

The scope of a variable specifies where the variable is defined. The availability of a variable within a program is determined by its scope. TypeScript variables can be of the following scopes

* **Global Scope** − Global variables are declared outside the programming constructs. These variables can be accessed from anywhere within your code.
* **Class Scope** − These variables are also called **fields**. Fields or class variables are declared within the class but outside the methods. These variables can be accessed using the object of the class. Fields can also be static. Static fields can be accessed using the class name.
* **Local Scope** − Local variables, as the name suggests, are declared within the constructs like methods, loops etc. Local variables are accessible only within the construct where they are declared.

var global\_num =12//global variable

classNumbers{

num\_val =13;//class variable

static sval =10;//static field

storeNum():void{

var local\_num =14;//local variable

}

}

console.log("Global num: "+global\_num)

console.log(Numbers.sval)//static variable

var obj =newNumbers();

console.log("Global num: "+obj.num\_val)

TypeScript Operators

The major operators in TypeScript can be classified as −

* Arithmetic operators
* Logical operators
* Relational operators
* Bitwise operators
* Assignment operators
* Ternary/conditional operator
* Increment/Decrement
* Miscellaneous (type of, instance of)

TypeScript - Decision Making

Decision-making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Shown below is the general form of a typical decision-making structure found in most of the programming languages −

A decision-making construct evaluates a condition before the instructions are executed. Decision-making constructs in TypeScript are classified as follows −

|  |  |
| --- | --- |
| **S.No.** | **Statement & Description** |
| 1. | [**if statement**](https://www.tutorialspoint.com/typescript/typescript_if_statement.htm)  An ‘if’ statement consists of a Boolean expression followed by one or more statements. |
| 2. | [**if...else statement**](https://www.tutorialspoint.com/typescript/typescript_if_else_statement.htm)  An ‘if’ statement can be followed by an optional ‘else’ statement, which executes when the Boolean expression is false. |
| 3. | [**else…if and nested if statements**](https://www.tutorialspoint.com/typescript/typescript_nested_if_statements.htm)  You can use one ‘if’ or ‘else if’ statement inside another ‘if’ or ‘else if’ statement(s). |
| 4. | [**switch statement**](https://www.tutorialspoint.com/typescript/typescript_switch_statement.htm)  A ‘switch’ statement allows a variable to be tested against a list of values. |

Loops

* While
* Do-while
* For

# **TypeScript – Functions**

* function function\_name (param1[:type], param2[:type], param3[:type])

### **Optional Parameters**

function disp\_details(id:number,name:string,mail\_id?:string){

console.log("ID:", id);

console.log("Name",name);

if(mail\_id!=undefined)

console.log("Email Id",mail\_id);

}

disp\_details(123,"John");

disp\_details(111,"mary","mary@xyz.com");

* The above example declares a parameterized function. Here, the third parameter, i.e., mail\_id is an optional parameter.
* If an optional parameter is not passed a value during the function call, the parameter’s value is set to undefined.
* The function prints the value of mail\_id only if the argument is passed a value.

## Rest Parameters

* Rest parameters are similar to variable arguments in Java

functionaddNumbers(...nums:number[]){

var i;

varsum:number=0;

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

sum = sum + nums[i];

}

console.log("sum of the numbers",sum)

}

addNumbers(1,2,3)

addNumbers(10,10,10,10,10)

## Default Parameters

* Function parameters can also be assigned values by default. However, such parameters can also be explicitly passed values.

function function\_name(param1[:type],param2[:type] = default\_value) {

}

## Anonymous Function

var res =function(a:number,b:number){

return a\*b;

};

console.log(res(12,2))

TypeScript and Object Orientation

TypeScript is Object-Oriented JavaScript. Object Orientation is a software development paradigm that follows real-world modelling. Object Orientation considers a program as a collection of objects that communicate with each other via mechanism called methods. TypeScript supports these object oriented components too.

* **Object** − An object is a real time representation of any entity. According to Grady Brooch, every object must have three features −
  + **State** − described by the attributes of an object
  + **Behavior** − describes how the object will act
  + **Identity** − a unique value that distinguishes an object from a set of similar such objects.
* **Class** − A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.
* **Method** − Methods facilitate communication between objects.

**Example: TypeScript and Object Orientation**

classGreeting{

greet():void{

console.log("Hello World!!!")

}

}

var obj =newGreeting();

obj.greet();

The above example defines a class *Greeting*. The class has a method *greet ()*. The method prints the string “Hello World” on the terminal. The **new** keyword creates an object of the class (obj). The object invokes the method *greet ()*.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

varGreeting=(function(){

functionGreeting(){

}

Greeting.prototype.greet=function(){

console.log("Hello World!!!");

};

returnGreeting;

}());

var obj =newGreeting();

obj.greet()

The output of the above program is given below −

Hello World!!!

The following example illustrates variable scopes in TypeScript.

Example: Variable Scope

var global\_num =12//global variable

classNumbers{

num\_val =13;//class variable

static sval =10;//static field

storeNum():void{

var local\_num =14;//local variable

}

}

console.log("Global num: "+global\_num)

console.log(Numbers.sval)//static variable

var obj =newNumbers();

console.log("Global num: "+obj.num\_val)

On transpiling, the following JavaScript code is generated −

var global\_num =12;//global variable

varNumbers=(function(){

functionNumbers(){

this.num\_val =13;//class variable

}

Numbers.prototype.storeNum=function(){

var local\_num =14;//local variable

};

Numbers.sval =10;//static field

returnNumbers;

}());

console.log("Global num: "+ global\_num);

console.log(Numbers.sval);//static variable

var obj =newNumbers();

console.log("Global num: "+ obj.num\_val);

It will produce the following output −

Global num: 12

10

Global num: 13

If you try accessing the local variable outside the method, it results in a compilation error.

error TS2095: Could not find symbol 'local\_num'.

TypeScript - Loops

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages.

TypeScript provides different types of loops to handle looping requirements. The following figure illustrates the classification of loops −

Definite Loop

A loop whose number of iterations are definite/fixed is termed as a **definite loop**. The *for loop* is an implementation of a definite loop.

|  |  |
| --- | --- |
| **S.No.** | **Loops & Description** |
| 1. | [**for loop**](https://www.tutorialspoint.com/typescript/typescript_for_loop.htm)  The for loop is an implementation of a definite loop. |

Indefinite Loop

An indefinite loop is used when the number of iterations in a loop is indeterminate or unknown.

Indefinite loops can be implemented using −

|  |  |
| --- | --- |
| **S.No** | **Loops & Description** |
| 1. | [**while loop**](https://www.tutorialspoint.com/typescript/typescript_while_loop.htm)  The while loop executes the instructions each time the condition specified evaluates to true. |
| 2. | [**do… while**](https://www.tutorialspoint.com/typescript/typescript_do_while_loop.htm)  The do…while loop is similar to the while loop except that the do...while loop doesn’t evaluate the condition for the first time the loop executes. |

Example: while versus do..while

varn:number=5

while(n >5){

console.log("Entered while")

}

do{

console.log("Entered do…while")

}while(n>5)

The example initially declares a while loop. The loop is entered only if the expression passed to while evaluates to true. In this example, the value of n is not greater than zero, hence the expression returns false and the loop is skipped.

On the other hand, the do…while loop executes statement once. This is because the initial iteration does not consider the Boolean expression. However, for the subsequent iteration, the while checks the condition and takes the control out of the loop.

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

var n =5;

while(n >5){

console.log("Entered while");

}

do{

console.log("Entered do…while");

}while(n >5);

The above code will produce the following output −

Entered do…while

The break Statement

The **break** statement is used to take the control out of a construct. Using **break** in a loop causes the program to exit the loop. Its syntax is as follows −

Syntax

break

Flow diagram

Example

Now, take a look at the following example code −

vari:number=1

while(i<=10){

if(i %5==0){

console.log ("The first multiple of 5 between 1 and 10 is : "+i)

break//exit the loop if the first multiple is found

}

i++

}//outputs 5 and exits the loop

On compiling, it will generate the following JavaScript code −

//Generated by typescript 1.8.10

var i =1;

while(i <=10){

if(i %5==0){

console.log("The first multiple of 5 between 1 and 10 is : "+ i);

break;//exit the loop if the first multiple is found

}

i++;

}//outputs 5 and exits the loop

It will produce the following output −

The first multiple of 5 between 1 and 10 is : 5

The continue Statement

The **continue** statement skips the subsequent statements in the current iteration and takes the control back to the beginning of the loop. Unlike the break statement, the continue doesn’t exit the loop. It terminates the current iteration and starts the subsequent iteration.

Syntax

continue

Flowchart

Example

An example of the continue statement is given below −

varnum:number=0

varcount:number=0;

for(num=0;num<=20;num++){

if(num %2==0){

continue

}

count++

}

console.log (" The count of odd values between 0 and 20 is: "+count)//outputs 10

The above example displays the number of even values between 0 and 20. The loop exits the current iteration if the number is even. This is achieved using the **continue** statement.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var num =0;

var count =0;

for(num =0; num <=20; num++){

if(num %2==0){

continue;

}

count++;

}

console.log(" The count of odd values between 0 and 20 is: "+ count);//outputs 10

Output

The count of odd values between 0 and 20 is: 10

The Infinite Loop

An infinite loop is a loop that runs endlessly. The **for** loop and the **while** loop can be used to make an endless loop.

Syntax: Infinite Loop using for loop

for(;;) {

//statements

}

Example: Infinite loop using for loop

for(;;){

console.log(“Thisis an endless loop”)

}

Syntax: Infinite loop using while loop

while(true) {

//statements

}

Example: Infinite loop using while loop

while(true){

console.log(“Thisis an endless loop”)

}

TypeScript - Functions

Functions are the building blocks of readable, maintainable, and reusable code. A function is a set of statements to perform a specific task. Functions organize the program into logical blocks of code. Once defined, functions may be called to access code. This makes the code reusable. Moreover, functions make it easy to read and maintain the program’s code.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

|  |  |
| --- | --- |
| **Sr.No** | **Funtions & Description** |
| 1. | [**Defining a Function**](https://www.tutorialspoint.com/typescript/typescript_defining_function.htm)  A function definition specifies what and how a specific task would be done. |
| 2. | [**Calling a Function**](https://www.tutorialspoint.com/typescript/typescript_calling_function.htm)  A function must be called so as to execute it. |
| 3. | [**Returning Functions**](https://www.tutorialspoint.com/typescript/typescript_returning_function.htm)  Functions may also return value along with control, back to the caller. |
| 4. | [**Parameterized Function**](https://www.tutorialspoint.com/typescript/typescript_parameterized_function.htm)  Parameters are a mechanism to pass values to functions. |

Optional Parameters

Optional parameters can be used when arguments need not be compulsorily passed for a function’s execution. A parameter can be marked optional by appending a question mark to its name. The optional parameter should be set as the last argument in a function. The syntax to declare a function with optional parameter is as given below −

function function\_name (param1[:type], param2[:type], param3[:type])

Example: Optional Parameters

function disp\_details(id:number,name:string,mail\_id?:string){

console.log("ID:", id);

console.log("Name",name);

if(mail\_id!=undefined)

console.log("Email Id",mail\_id);

}

disp\_details(123,"John");

disp\_details(111,"mary","mary@xyz.com");

* The above example declares a parameterized function. Here, the third parameter, i.e., mail\_id is an optional parameter.
* If an optional parameter is not passed a value during the function call, the parameter’s value is set to undefined.
* The function prints the value of mail\_id only if the argument is passed a value.

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

function disp\_details(id, name, mail\_id){

console.log("ID:", id);

console.log("Name", name);

if(mail\_id !=undefined)

console.log("Email Id", mail\_id);

}

disp\_details(123,"John");

disp\_details(111,"mary","mary@xyz.com");

The above code will produce the following output −

ID:123

Name John

ID: 111

Name mary

Email Id mary@xyz.com

Rest Parameters

Rest parameters are similar to variable arguments in Java. Rest parameters don’t restrict the number of values that you can pass to a function. However, the values passed must all be of the same type. In other words, rest parameters act as placeholders for multiple arguments of the same type.

To declare a rest parameter, the parameter name is prefixed with three periods. Any nonrest parameter should come before the rest parameter.

Example: Rest Parameters

functionaddNumbers(...nums:number[]){

var i;

varsum:number=0;

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

sum = sum + nums[i];

}

console.log("sum of the numbers",sum)

}

addNumbers(1,2,3)

addNumbers(10,10,10,10,10)

* The function addNumbers() declaration, accepts a rest parameter *nums*. The rest parameter’s data type must be set to an array. Moreover, a function can have at the most one rest parameter.
* The function is invoked twice, by passing three and six values, respectively.
* The for loop iterates through the argument list, passed to the function and calculates their sum.

On compiling, it will generate following JavaScript code −

functionaddNumbers(){

var nums =[];

for(var \_i =0; \_i <arguments.length; \_i++){

nums[\_i -0]= arguments[\_i];

}

var i;

var sum =0;

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

sum = sum + nums[i];

}

console.log("sum of the numbers", sum);

}

addNumbers(1,2,3);

addNumbers(10,10,10,10,10);

The output of the above code is as follows −

sum of numbers 6

sum of numbers 50

Default Parameters

Function parameters can also be assigned values by default. However, such parameters can also be explicitly passed values.

Syntax

function function\_name(param1[:type],param2[:type] = default\_value) {

}

**Note** − A parameter cannot be declared optional and default at the same time.

Example: Default parameters

function calculate\_discount(price:number,rate:number =0.50){

var discount = price \* rate;

console.log("Discount Amount: ",discount);

}

calculate\_discount(1000)

calculate\_discount(1000,0.30)

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

function calculate\_discount(price, rate){

if(rate ===void0){ rate=0.50;}

var discount = price \* rate;

console.log("Discount Amount: ", discount);

}

calculate\_discount(1000);

calculate\_discount(1000,0.30);

Its output is as follows −

Discount amount : 500

Discount amount : 300

* The example declares the function, *calculate\_discount*. The function has two parameters - price and rate.
* The value of the parameter *rate* is set to *0.50* by default.
* The program invokes the function, passing to it only the value of the parameter price. Here, the value of *rate* is *0.50* (default)
* The same function is invoked, but with two arguments. The default value of *rate* is overwritten and is set to the value explicitly passed.

Anonymous Function

Functions that are not bound to an identifier (function name) are called as **anonymous functions**. These functions are dynamically declared at runtime. Anonymous functions can accept inputs and return outputs, just as standard functions do. An anonymous function is usually not accessible after its initial creation.

Variables can be assigned an anonymous function. Such an expression is called a function expression.

Syntax

var res = function( [arguments] ) { ... }

Example ─ A Simple Anonymous function

var msg =function(){

return"hello world";

}

console.log(msg())

On compiling, it will generate the same code in JavaScript.

It will produce the following output −

hello world

Example ─ Anonymous function with parameters

var res =function(a:number,b:number){

return a\*b;

};

console.log(res(12,2))

The anonymous function returns the product of the values passed to it.

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

var res =function(a, b){

return a \* b;

};

console.log(res(12,2));

The output of the above code is as follows −

24

Function Expression and Function Declaration ─ Are they synonymous?

Function expression and function declaration are not synonymous. Unlike a function expression, a function declaration is bound by the function name.

The fundamental difference between the two is that, function declarations are parsed before their execution. On the other hand, function expressions are parsed only when the script engine encounters it during execution.

When the JavaScript parser sees a function in the main code flow, it assumes Function Declaration. When a function comes as a part of a statement, it is a Function Expression.

The Function Constructor

TypeScript also supports defining a function with the built-in JavaScript constructor called Function (). TypeScript also supports defining a function with the built-in JavaScript constructor called Function ().

Syntax

var res = new Function( [arguments] ) { ... }.

Example

var myFunction =newFunction("a","b","return a \* b");

var x =myFunction(4,3);

console.log(x);

The new Function() is a call to the constructor which in turn creates and returns a function reference.

On compiling, it will generate the same code in JavaScript.

The output of the above example code is as follows −

12

Recursion and TypeScript Functions

Recursion is a technique for iterating over an operation by having a function call to itself repeatedly until it arrives at a result. Recursion is best applied when you need to call the same function repeatedly with different parameters from within a loop.

Example – Recursion

function factorial(number){

if(number <=0){// termination case

return1;

}else{

return(number \*factorial(number -1));// function invokes itself

}

};

console.log(factorial(6));// outputs 720

On compiling, it will generate the same code in JavaScript.

Here is its output −

720

Example: Anonymous Recursive Function

(function(){

var x ="Hello!!";

console.log(x)

})()// the function invokes itself using a pair of parentheses ()

On compiling, it will generate the same code in JavaScript.

Its output is as follows −

Hello!!

Lambda Functions

Lambda refers to anonymous functions in programming. Lambda functions are a concise mechanism to represent anonymous functions. These functions are also called as **Arrow functions**.

Lambda Function - Anatomy

There are 3 parts to a Lambda function −

* **Parameters** − A function may optionally have parameters
* **The fat arrow notation/lambda notation (=>)** − It is also called as the goes to operator
* **Statements** − represent the function’s instruction set

**Tip** − By convention, the use of single letter parameter is encouraged for a compact and precise function declaration.

Lambda Expression

It is an anonymous function expression that points to a single line of code. Its syntax is as follows −

( [param1, parma2,…param n] )=>statement;

Example: Lambda Expression

var foo =(x:number)=>10+ x

console.log(foo(100))//outputs 110

The program declares a lambda expression function. The function returns the sum of 10 and the argument passed.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var foo =function(x){return10+ x;};

console.log(foo(100));//outputs 110

Here is the output of the above code −

110

Lambda Statement

Lambda statement is an anonymous function declaration that points to a block of code. This syntax is used when the function body spans multiple lines. Its syntax is as follows −

( [param1, parma2,…param n] )=> {

//code block

}

Example: Lambda statement

var foo =(x:number)=>{

x =10+ x

console.log(x)

}

foo(100)

The function’s reference is returned and stored in the variable **foo**.

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

var foo =function(x){

x =10+ x;

console.log(x);

};

foo(100);

The output of the above program is as follows −

110

Syntactic Variations

Parameter type Inference

It is not mandatory to specify the data type of a parameter. In such a case the data type of the parameter is any. Let us take a look at the following code snippet −

var func =(x)=>{

if(typeof x=="number"){

console.log(x+" is numeric")

}elseif(typeof x=="string"){

console.log(x+" is a string")

}

}

func(12)

func("Tom")

On compiling, it will generate the following JavaScript code −

//Generated by typescript 1.8.10

var func =function(x){

if(typeof x =="number"){

console.log(x +" is numeric");

}elseif(typeof x =="string"){

console.log(x +" is a string");

}

};

func(12);

func("Tom");

Its output is as follows −

12 is numeric

Tom is a string

Optional parentheses for a single parameter

var display = x=>{

console.log("The function got "+x)

}

display(12)

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

var display =function(x){

console.log("The function got "+ x);

};

display(12);

Its output is as follows −

The function got 12

Optional braces for a single statement, Empty parentheses for no parameter

The following example shows these two Syntactic variations.

var disp =()=>{

console.log("Function invoked");

}

disp();

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

var disp =function(){

console.log("Function invoked");

};

disp();

Its output is as follows −

Function invoked

Function Overloads

Functions have the capability to operate differently on the basis of the input provided to them. In other words, a program can have multiple methods with the same name with different implementation. This mechanism is termed as Function Overloading. TypeScript provides support for function overloading.

To overload a function in TypeScript, you need to follow the steps given below −

**Step 1** − Declare multiple functions with the same name but different function signature. Function signature includes the following.

* **The data type of the parameter**

function disp(string):void;

function disp(number):void;

* **The number of parameters**

function disp(n1:number):void;

function disp(x:number,y:number):void;

* **The sequence of parameters**

function disp(n1:number,s1:string):void;

function disp(s:string,n:number):void;

**Note** − The function signature doesn’t include the function’s return type.

**Step 2** − The declaration must be followed by the function definition. The parameter types should be set to **any** if the parameter types differ during overload. Additionally, for **case b** explained above, you may consider marking one or more parameters as optional during the function definition.

**Step 3** − Finally, you must invoke the function to make it functional.

Example

Let us now take a look at the following example code −

function disp(s1:string):void;

function disp(n1:number,s1:string):void;

function disp(x:any,y?:any):void{

console.log(x);

console.log(y);

}

disp("abc")

disp(1,"xyz");

* The first two lines depict the function overload declaration. The function has two overloads −
  + Function that accepts a single string parameter.
  + Function that accepts two values of type number and string respectively.
* The third line defines the function. The data type of the parameters are set to **any**. Moreover, the second parameter is optional here.
* The overloaded function is invoked by the last two statements.

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

functiondisp(x, y){

console.log(x);

console.log(y);

}

disp("abc");

disp(1,"xyz");

The above code will produce the following output −

abc

1

xyz

TypeScript - Numbers

TypeScript like JavaScript supports numeric values as Number objects. A number object converts numeric literal to an instance of the number class. The Number class acts as a wrapper and enables manipulation of numeric literals as they were objects.

Syntax

var var\_name = new Number(value)

In case a non-numeric argument is passed as an argument to the Number’s constructor, it returns NaN (Not–a–Number)

The following table lists a set of properties of the Number object −

|  |  |
| --- | --- |
| **S.No.** | **Property & Description** |
| 1. | **MAX\_VALUE**  The largest possible value a number in JavaScript can have 1.7976931348623157E+308. |
| 2. | **MIN\_VALUE**  The smallest possible value a number in JavaScript can have 5E-324. |
| 3. | **NaN**  Equal to a value that is not a number. |
| 4. | **NEGATIVE\_INFINITY**  A value that is less than MIN\_VALUE. |
| 5. | **POSITIVE\_INFINITY**  A value that is greater than MAX\_VALUE. |
| 6. | **prototype**  A static property of the Number object. Use the prototype property to assign new properties and methods to the Number object in the current document. |
| 7. | **constructor**  Returns the function that created this object's instance. By default, this is the Number object. |

Example

console.log("TypeScript Number Properties: ");

console.log("Maximum value that a number variable can hold: "+Number.MAX\_VALUE);

console.log("The least value that a number variable can hold: "+Number.MIN\_VALUE);

console.log("Value of Negative Infinity: "+Number.NEGATIVE\_INFINITY);

console.log("Value of Negative Infinity:"+Number.POSITIVE\_INFINITY);

On compiling, it will generate the same code in JavaScript.

Its output is as follows −

TypeScript Number Properties:

Maximum value that a number variable can hold: 1.7976931348623157e+308

The least value that a number variable can hold: 5e-324

Value of Negative Infinity: -Infinity

Value of Negative Infinity:Infinity

Example: NaN

var month =0

if( month<=0|| month >12){

month =Number.NaN

console.log("Month is "+ month)

}else{

console.log("Value Accepted..")

}

On compiling, it will generate the same code in JavaScript.

Its output is as follows −

Month is NaN

Example: prototye

function employee(id:number,name:string){

this.id = id

this.name = name

}

var emp =new employee(123,"Smith")

employee.prototype.email ="smith@abc.com"

console.log("Employee 's Id: "+emp.id)

console.log("Employee's name: "+emp.name)

console.log("Employee's Email ID: "+emp.email)

On compiling, it will generate the following JavaScript code −

//Generated by typescript 1.8.10

functionemployee(id, name){

this.id = id;

this.name = name;

}

var emp =newemployee(123,"Smith");

employee.prototype.email ="smith@abc.com";

console.log("Employee 's Id: "+ emp.id);

console.log("Employee's name: "+ emp.name);

console.log("Employee's Email ID: "+ emp.email);

Its output is as follows −

Employee’s Id: 123

Emaployee’s name: Smith

Employee’s Email ID: smith@abc.com

Number Methods

The Number object contains only the default methods that are a part of every object's definition. Some of the commonly used methods are listed below −

|  |  |
| --- | --- |
| **S.No.** | **Methods & Description** |
| 1. | [**toExponential()**](https://www.tutorialspoint.com/typescript/typescript_number_toexponential.htm)  Forces a number to display in exponential notation, even if the number is in the range in which JavaScript normally uses standard notation. |
| 2. | [**toFixed()**](https://www.tutorialspoint.com/typescript/typescript_number_tofixed.htm)  Formats a number with a specific number of digits to the right of the decimal. |
| 3. | [**toLocaleString()**](https://www.tutorialspoint.com/typescript/typescript_number_tolocalestring.htm)  Returns a string value version of the current number in a format that may vary according to a browser's local settings. |
| 4. | [**toPrecision()**](https://www.tutorialspoint.com/typescript/typescript_number_toprecision.htm)  Defines how many total digits (including digits to the left and right of the decimal) to display of a number. A negative precision will throw an error. |
| 5. | [**toString()**](https://www.tutorialspoint.com/typescript/typescript_number_tostring.htm)  Returns the string representation of the number's value. The function is passed the radix, an integer between 2 and 36 specifying the base to use for representing numeric values. |
| 6. | [**valueOf()**](https://www.tutorialspoint.com/typescript/typescript_number_valueof.htm)  Returns the number's primitive value. |

TypeScript - Strings

The String object lets you work with a series of characters. It wraps the string primitive data type with a number of helper methods.

Syntax

var var\_name = new String(string);

A list of the methods available in String object along with their description is given below −

|  |  |
| --- | --- |
| **S.No.** | **Property & Description** |
| 1. | [**Constructor**](https://www.tutorialspoint.com/typescript/typescript_string_constructor.htm)  Returns a reference to the String function that created the object. |
| 2. | [**Length**](https://www.tutorialspoint.com/typescript/typescript_string_length.htm)  Returns the length of the string. |
| 3. | [**Prototype**](https://www.tutorialspoint.com/typescript/typescript_object_prototype.htm)  The prototype property allows you to add properties and methods to an object. |

String Methods

A list of the methods available in String object along with their description is given below −

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1. | [**charAt()**](https://www.tutorialspoint.com/typescript/typescript_string_charat.htm)  Returns the character at the specified index. |
| 2. | [**charCodeAt()**](https://www.tutorialspoint.com/typescript/typescript_string_charcodeat.htm)  Returns a number indicating the Unicode value of the character at the given index. |
| 3. | [**concat()**](https://www.tutorialspoint.com/typescript/typescript_string_concat.htm)  Combines the text of two strings and returns a new string. |
| 4. | [**indexOf()**](https://www.tutorialspoint.com/typescript/typescript_string_indexof.htm)  Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found. |
| 5. | [**lastIndexOf()**](https://www.tutorialspoint.com/typescript/typescript_string_lastindexof.htm)  Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found. |
| 6. | [**localeCompare()**](https://www.tutorialspoint.com/typescript/typescript_string_localecompare.htm)  Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order. |
| 7. | **match()**  Used to match a regular expression against a string. |

|  |  |
| --- | --- |
| 8. | [**replace()**](https://www.tutorialspoint.com/typescript/typescript_string_replace.htm)  Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring. |
| 9. | [**search()**](https://www.tutorialspoint.com/typescript/typescript_string_search.htm)  Executes the search for a match between a regular expression and a specified string. |
| 10. | [**slice()**](https://www.tutorialspoint.com/typescript/typescript_string_slice.htm)  Extracts a section of a string and returns a new string. |
| 11. | [**split()**](https://www.tutorialspoint.com/typescript/typescript_string_split.htm)  Splits a String object into an array of strings by separating the string into substrings. |
| 12. | [**substr()**](https://www.tutorialspoint.com/typescript/typescript_string_substr.htm)  Returns the characters in a string beginning at the specified location through the specified number of characters. |
| 13. | [**substring()**](https://www.tutorialspoint.com/typescript/typescript_string_substring.htm)  Returns the characters in a string between two indexes into the string. |
| 14. | [**toLocaleLowerCase()**](https://www.tutorialspoint.com/typescript/typescript_string_tolocalelowercase.htm)  The characters within a string are converted to lower case while respecting the current locale. |
| 15. | [**toLocaleUpperCase()**](https://www.tutorialspoint.com/typescript/typescript_string_tolocaleuppercase.htm)  The characters within a string are converted to upper case while respecting the current locale. |
| 16. | [**toLowerCase()**](https://www.tutorialspoint.com/typescript/typescript_string_tolowercase.htm)  Returns the calling string value converted to lower case. |
| 17. | [**toString()**](https://www.tutorialspoint.com/typescript/typescript_string_tostring.htm)  Returns a string representing the specified object. |
| 18. | [**toUpperCase()**](https://www.tutorialspoint.com/typescript/typescript_string_touppercase.htm)  Returns the calling string value converted to uppercase. |
| 19. | [**valueOf()**](https://www.tutorialspoint.com/typescript/typescript_string_valueof.htm)  Returns the primitive value of the specified object. |

TypeScript - Arrays

The use of variables to store values poses the following limitations −

* Variables are scalar in nature. In other words, a variable declaration can only contain a single at a time. This means that to store n values in a program n variable declarations will be needed. Hence, the use of variables is not feasible when one needs to store a larger collection of values.
* Variables in a program are allocated memory in random order, thereby making it difficult to retrieve/read the values in the order of their declaration.

TypeScript introduces the concept of arrays to tackle the same. An array is a homogenous collection of values. To simplify, an array is a collection of values of the same data type. It is a user defined type.

Features of an Array

Here is a list of the features of an array −

* An array declaration allocates sequential memory blocks.
* Arrays are static. This means that an array once initialized cannot be resized.
* Each memory block represents an array element.
* Array elements are identified by a unique integer called as the subscript / index of the element.
* Like variables, arrays too, should be declared before they are used. Use the var keyword to declare an array.
* Array initialization refers to populating the array elements.
* Array element values can be updated or modified but cannot be deleted.

Declaring and Initializing Arrays

To declare an initialize an array in Typescript use the following syntax −

Syntax

var array\_name[:datatype]; //declaration

array\_name = [val1,val2,valn..] //initialization

An array declaration without the data type is deemed to be of the type any. The type of such an array is inferred from the data type of the array’s first element during initialization.

For example, a declaration like − **var numlist:number[] = [2,4,6,8]** will create an array as given below −

The array pointer refers to the first element by default.

**Arrays may be declared and initialized in a single statement**. The syntax for the same is −

var array\_name[:data type] = [val1,val2…valn]

**Note** − The pair of [] is called the dimension of the array.

Accessing Array Elements

The array name followed by the subscript is used refer to an array element. Its syntax is as follows −

array\_name[subscript] = value

Example: Simple Array

varalphas:string[];

alphas =["1","2","3","4"]

console.log(alphas[0]);

console.log(alphas[1]);

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

var alphas;

alphas =["1","2","3","4"];

console.log(alphas[0]);

console.log(alphas[1]);

The output of the above code is as follows −

1

2

Example: Single statement declaration and initialization

varnums:number[]=[1,2,3,3]

console.log(nums[0]);

console.log(nums[1]);

console.log(nums[2]);

console.log(nums[3]);

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

var nums =[1,2,3,3];

console.log(nums[0]);

console.log(nums[1]);

console.log(nums[2]);

console.log(nums[3]);

Its output is as follows −

1

2

3

3

Array Object

An array can also be created using the Array object. The Array constructor can be passed.

* A numeric value that represents the size of the array or
* A list of comma separated values.

The following example shows how to create an array using this method.

Example

var arr\_names:number[]=newArray(4)

for(var i =0;i<arr\_names.length;i++;){

arr\_names[i]= i \*2

console.log(arr\_names[i])

}

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var arr\_names =newArray(4);

for(var i =0; i < arr\_names.length; i++){

arr\_names[i]= i \*2;

console.log(arr\_names[i]);

}

Its output is as follows −

0

2

4

6

Example: Array Constructor accepts comma separated values

varnames:string[]=newArray("Mary","Tom","Jack","Jill")

for(var i =0;i<names.length;i++){

console.log(names[i])

}

On compiling, it will generate following JavaScript code −

//Generated by typescript 1.8.10

var names =newArray("Mary","Tom","Jack","Jill");

for(var i =0; i <names.length; i++){

console.log(names[i]);

}

Its output is as follows −

Mary

Tom

Jack

Jill

Array Methods

A list of the methods of the Array object along with their description is given below.

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1. | [**concat()**](https://www.tutorialspoint.com/typescript/typescript_array_concat.htm)  Returns a new array comprised of this array joined with other array(s) and/or value(s). |
| 2. | [**every()**](https://www.tutorialspoint.com/typescript/typescript_array_every.htm)  Returns true if every element in this array satisfies the provided testing function. |
| 3. | [**filter()**](https://www.tutorialspoint.com/typescript/typescript_array_filter.htm)  Creates a new array with all of the elements of this array for which the provided filtering function returns true. |
| 4. | [**forEach()**](https://www.tutorialspoint.com/typescript/typescript_array_foreach.htm)  Calls a function for each element in the array. |
| 5. | [**indexOf()**](https://www.tutorialspoint.com/typescript/typescript_array_indexof.htm)  Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found. |
| 6. | [**join()**](https://www.tutorialspoint.com/typescript/typescript_array_join.htm)  Joins all elements of an array into a string. |
| 7. | [**lastIndexOf()**](https://www.tutorialspoint.com/typescript/typescript_array_lastindexof.htm)  Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found. |
| 8. | [**map()**](https://www.tutorialspoint.com/typescript/typescript_array_map.htm)  Creates a new array with the results of calling a provided function on every element in this array. |
| 9. | [**pop()**](https://www.tutorialspoint.com/typescript/typescript_array_pop.htm)  Removes the last element from an array and returns that element. |
| 10. | [**push()**](https://www.tutorialspoint.com/typescript/typescript_array_push.htm)  Adds one or more elements to the end of an array and returns the new length of the array. |
| 11. | [**reduce()**](https://www.tutorialspoint.com/typescript/typescript_array_reduce.htm)  Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value. |
| 12. | [**reduceRight()**](https://www.tutorialspoint.com/typescript/typescript_array_reduceright.htm)  Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value. |
| 13. | [**reverse()**](https://www.tutorialspoint.com/typescript/typescript_array_reverse.htm)  Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first. |
| 14. | [**shift()**](https://www.tutorialspoint.com/typescript/typescript_array_shift.htm)  Removes the first element from an array and returns that element. |
| 15. | [**slice()**](https://www.tutorialspoint.com/typescript/typescript_array_slice.htm)  Extracts a section of an array and returns a new array. |
| 16. | [**some()**](https://www.tutorialspoint.com/typescript/typescript_array_some.htm)  Returns true if at least one element in this array satisfies the provided testing function. |
| 17. | [**sort()**](https://www.tutorialspoint.com/typescript/typescript_array_sort.htm)  Sorts the elements of an array. |
| 18. | [**splice()**](https://www.tutorialspoint.com/typescript/typescript_array_splice.htm)  Adds and/or removes elements from an array. |
| 19. | [**toString()**](https://www.tutorialspoint.com/typescript/typescript_array_tostring.htm)  Returns a string representing the array and its elements. |
| 20. | [**unshift()**](https://www.tutorialspoint.com/typescript/typescript_array_unshift.htm)  Adds one or more elements to the front of an array and returns the new length of the array. |

Array Destructuring

Refers to breaking up the structure of an entity. TypeScript supports destructuring when used in the context of an array.

Example

vararr:number[]=[12,13]

var[x,y]= arr

console.log(x)

console.log(y)

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var arr =[12,13];

var x =arr[0], y = arr[1];

console.log(x);

console.log(y);

Its output is as follows −

12

13

Array Traversal using for…in loop

One can use the **for…in** loop to traverse through an array.

varj:any;

varnums:number[]=[1001,1002,1003,1004]

for(j in nums){

console.log(nums[j])

}

The loop performs an index based array traversal.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var j;

var nums =[1001,1002,1003,1004];

for(j in nums){

console.log(nums[j]);

}

The output of the above code is given below −

1001

1002

1003

1004

Arrays in TypeScript

TypeScript supports the following concepts in arrays −

|  |  |
| --- | --- |
| **S.No.** | **Concept & Description** |
| 1. | [**Multi-dimensional arrays**](https://www.tutorialspoint.com/typescript/typescript_multi_dimensional_arrays.htm)  TypeScript supports multidimensional arrays. The simplest form of the multidimensional array is the twodimensional array. |
| 2. | [**Passing arrays to functions**](https://www.tutorialspoint.com/typescript/typescript_passing_arrays_to_functions.htm)  You can pass to the function a pointer to an array by specifying the array's name without an index. |
| 3. | [**Return array from functions**](https://www.tutorialspoint.com/typescript/typescript_return_array_from_functions.htm)  Allows a function to return an array |

TypeScript - Tuples

At times, there might be a need to store a collection of values of varied types. Arrays will not serve this purpose. TypeScript gives us a data type called tuple that helps to achieve such a purpose.

It represents a heterogeneous collection of values. In other words, tuples enable storing multiple fields of different types. Tuples can also be passed as parameters to functions.

Syntax

var tuple\_name = [value1,value2,value3,…value n]

For Example

var mytuple =[10,"Hello"];

You can also declare an empty tuple in Typescript and choose to initialize it later.

var mytuple =[];

mytuple[0]=120

mytuple[1]=234

Accessing values in Tuples

Tuple values are individually called items. Tuples are index based. This means that items in a tuple can be accessed using their corresponding numeric index. Tuple item’s index starts from zero and extends up to n-1(where n is the tuple’s size).

Syntax

tuple\_name[index]

Example: Simple Tuple

var mytuple =[10,"Hello"];//create a tuple

console.log(mytuple[0])

console.log(mytuple[1])

In the above example, a tuple, **mytuple**, is declared. The tuple contains values of numeric and string types respectively.

On compiling, it will generate the same code in JavaScript.

Its output is as follows −

10

Hello

Example: Empty Tuple

var tup =[]

tup[0]=12

tup[1]=23

console.log(tup[0])

console.log(tup[1])

On compiling, it will generate the same code in JavaScript.

Its output is as follows −

12

23

Tuple Operations

Tuples in TypeScript supports various operations like pushing a new item, removing an item from the tuple, etc.

Example

var mytuple =[10,"Hello","World","typeScript"];

console.log("Items before push "+mytuple.length)// returns the tuple size

mytuple.push(12)// append value to the tuple

console.log("Items after push "+mytuple.length)

console.log("Items before pop "+mytuple.length)

console.log(mytuple.pop()+" popped from the tuple")// removes and returns the last item

console.log("Items after pop "+mytuple.length)

* The push() appends an item to the tuple
* The pop() removes and returns the last value in the tuple

On compiling, it will generate the same code in JavaScript.

The output of the above code is as follows −

Items before push 4

Items after push 5

Items before pop 5

12 popped from the tuple

Items after pop 4

Updating Tuples

Tuples are mutable which means you can update or change the values of tuple elements.

Example

var mytuple =[10,"Hello","World","typeScript"];//create a tuple

console.log("Tuple value at index 0 "+mytuple[0])

//update a tuple element

mytuple[0]=121

console.log("Tuple value at index 0 changed to "+mytuple[0])

On compiling, it will generate the same code in JavaScript.

The output of the above code is as follows −

Tuple value at index 0 10

Tuple value at index 0 changed to 121

Destructuring a Tuple

Destructuring refers to breaking up the structure of an entity. TypeScript supports destructuring when used in the context of a tuple.

Example

var a =[10,"hello"]

var[b,c]= a

console.log( b)

console.log( c)

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var a =[10,"hello"];

var b =a[0], c = a[1];

console.log(b);

console.log(c);

Its output is as follows −

10

hello

TypeScript - Union

TypeScript 1.4 gives programs the ability to combine one or two types. Union types are a powerful way to express a value that can be one of the several types. Two or more data types are combined using the pipe symbol (|) to denote a Union Type. In other words, a union type is written as a sequence of types separated by vertical bars.

Syntax: Union literal

Type1|Type2|Type3

Example: Union Type Variable

varval:string|number

val =12

console.log("numeric value of val "+val)

val ="This is a string"

console.log("string value of val "+val)

In the above example, the variable’s type is union. It means that the variable can contain either a number or a string as its value.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var val;

val =12;

console.log("numeric value of val "+ val);

val ="This is a string";

console.log("string value of val "+ val);

Its output is as follows −

numeric value of val 12

string value of val this is a string

Example: Union Type and function parameter

function disp(name:string|string[]){

if(typeof name =="string"){

console.log(name)

}else{

var i;

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

console.log(name[i])

}

}

}

disp("mark")

console.log("Printing names array....")

disp(["Mark","Tom","Mary","John"])

The function disp() can accept argument either of the type string or a string array.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

function disp(name){

if(typeof name =="string"){

console.log(name);

}else{

var i;

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

console.log(name[i]);

}

}

}

disp("mark");

console.log("Printing names array....");

disp(["Mark","Tom","Mary","John"]);

The output is as follows −

Mark

Printing names array….

Mark

Tom

Mary

John

Union Type and Arrays

Union types can also be applied to arrays, properties and interfaces. The following illustrates the use of union type with an array.

Example: Union Type and Array

vararr:number[]|string[];

vari:number;

arr =[1,2,4]

console.log("\*\*numeric array\*\*")

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

console.log(arr[i])

}

arr =["Mumbai","Pune","Delhi"]

console.log("\*\*string array\*\*")

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

console.log(arr[i])

}

The program declares an array. The array can represent either a numeric collection or a string collection.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var arr;

var i;

arr =[1,2,4];

console.log("\*\*numeric array\*\*");

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

console.log(arr[i]);

}

arr =["Mumbai","Pune","Delhi"];

console.log("\*\*string array\*\*");

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

console.log(arr[i]);

}

Its output is as follows −

\*\*numeric array\*\*

1

2

4

\*\*string array\*\*

Mumbai

Pune

Delhi

TypeScript - Interfaces

An interface is a syntactical contract that an entity should conform to. In other words, an interface defines the syntax that any entity must adhere to.

Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Let’s consider an object −

var person ={

FirstName:"Tom",

LastName:"Hanks",

sayHi:()=>{return"Hi"}

};

If we consider the signature of the object, it could be −

{

FirstName:string,

LastName:string,

sayHi()=>string

}

To reuse the signature across objects we can define it as an interface.

Declaring Interfaces

The interface keyword is used to declare an interface. Here is the syntax to declare an interface −

Syntax

interface interface\_name {

}

Example: Interface and Objects

interfaceIPerson{

firstName:string,

lastName:string,

sayHi:()=>string

}

varcustomer:IPerson={

firstName:"Tom",

lastName:"Hanks",

sayHi:():string=>{return"Hi there"}

}

console.log("Customer Object ")

console.log(customer.firstName)

console.log(customer.lastName)

console.log(customer.sayHi())

varemployee:IPerson={

firstName:"Jim",

lastName:"Blakes",

sayHi:():string=>{return"Hello!!!"}

}

console.log("Employee Object ")

console.log(employee.firstName) console.log(employee.lastName)

The example defines an interface. The customer object is of the type IPerson. Hence, it will now be binding on the object to define all properties as specified by the interface.

Another object with following signature, is still considered as IPerson because that object is treated by its size or signature.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var customer ={ firstName:"Tom", lastName:"Hanks",

sayHi:function(){return"Hi there";}

};

console.log("Customer Object ");

console.log(customer.firstName);

console.log(customer.lastName);

console.log(customer.sayHi());

var employee ={ firstName:"Jim", lastName:"Blakes",

sayHi:function(){return"Hello!!!";}};

console.log("Employee Object ");

console.log(employee.firstName);

console.log(employee.lastName);

The output of the above example code is as follows −

Customer object

Tom

Hanks

Hi there

Employee object

Jim

Blakes

Hello!!!

Interfaces are not to be converted to JavaScript. It’s just part of TypeScript. If you see the screen shot of TS Playground tool there is no java script emitted when you declare an interface unlike a class. So interfaces have zero runtime JavaScript impact.

Union Type and Interface

The following example shows the use of Union Type and Interface −

interfaceRunOptions{

program:string;

commandline:string[]|string|(()=>string);

}

//commandline as string

varoptions:RunOptions={program:"test1",commandline:"Hello"};

console.log(options.commandline)

//commandline as a string array

options ={program:"test1",commandline:["Hello","World"]};

console.log(options.commandline[0]);

console.log(options.commandline[1]);

//commandline as a function expression

options ={program:"test1",commandline:()=>{return"\*\*Hello World\*\*";}};

varfn:any= options.commandline;

console.log(fn());

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

//commandline as string

var options ={ program:"test1", commandline:"Hello"};

console.log(options.commandline);

//commandline as a string array

options ={ program:"test1", commandline:["Hello","World"]};

console.log(options.commandline[0]);

console.log(options.commandline[1]);

//commandline as a function expression

options ={ program:"test1", commandline:function(){return"\*\*Hello World\*\*";}};

var fn =options.commandline;

console.log(fn());

Its output is as follows −

Hello

Hello

World

\*\*Hello World\*\*

Interfaces and Arrays

Interface can define both the kind of key an array uses and the type of entry it contains. Index can be of type string or type number.

Example

interface namelist {

[index:number]:string

}

var list2:namelist=["John",1,"Bran"]//Error. 1 is not type string

interface ages {

[index:string]:number

}

varagelist:ages;

agelist["John"]=15// Ok

agelist[2]="nine"// Error

Interfaces and Inheritance

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.

Use the extends keyword to implement inheritance among interfaces.

Syntax: Single Interface Inheritance

Child\_interface\_name extends super\_interface\_name

Syntax: Multiple Interface Inheritance

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

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

Example: Simple Interface Inheritance

interfacePerson{

age:number

}

interfaceMusicianextendsPerson{

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

interfaceIParent1{

v1:number

}

interfaceIParent2{

v2:number

}

interfaceChildextendsIParent1,IParent2{}

varIobj: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

varIobj={ 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

TypeScript - 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.

classPerson{

}

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

varPerson=(function(){

functionPerson(){

}

returnPerson;

}());

Example: Declaring a class

classCar{

//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

varCar=(function(){

//constructor

functionCar(engine){

this.engine= engine;

}

//function

Car.prototype.disp=function(){

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

};

returnCar;

}());

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 =newCar("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

classCar{

//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 =newCar("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

varCar=(function(){

//constructor

functionCar(engine){

this.engine= engine;

}

//function

Car.prototype.disp=function(){

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

};

returnCar;

}());

//create an object

var obj =newCar("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

Class Inheritance

TypeScript supports the concept of Inheritance. Inheritance is the ability of a program to create new classes from an existing class. The class that is extended to create newer classes is called the parent class/super class. The newly created classes are called the child/sub classes.

A class inherits from another class using the ‘extends’ keyword. Child classes inherit all properties and methods except constructors from the parent class.

Syntax

class child\_class\_name extends parent\_class\_name

However, TypeScript doesn’t support multiple inheritance.

Example: Class Inheritance

classShape{

Area:number

constructor(a:number){

this.Area= a

}

}

classCircleextendsShape{

disp():void{

console.log("Area of the circle: "+this.Area)

}

}

var obj =newCircle(223);

obj.disp()

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var \_\_extends =(this&&this.\_\_extends)||function(d, b){

for(var p in b)if(b.hasOwnProperty(p)) d[p]= b[p];

function \_\_(){this.constructor = d;}

d.prototype = b ===null?Object.create(b):(\_\_.prototype = b.prototype,new \_\_());

};

varShape=(function(){

functionShape(a){

this.Area= a;

}

returnShape;

}());

varCircle=(function(\_super){

\_\_extends(Circle, \_super);

functionCircle(){

\_super.apply(this, arguments);

}

Circle.prototype.disp=function(){

console.log("Area of the circle: "+this.Area);

};

returnCircle;

}(Shape));

var obj =newCircle(223);

obj.disp();

The output of the above code is as follows −

Area of the Circle: 223

The above example declares a class Shape. The class is extended by the Circle class. Since there is an inheritance relationship between the classes, the child class i.e. the class Car gets an implicit access to its parent class attribute i.e. area.

Inheritance can be classified as −

* **Single** − Every class can at the most extend from one parent class
* **Multiple** − A class can inherit from multiple classes. TypeScript doesn’t support multiple inheritance.
* **Multi-level** − The following example shows how multi-level inheritance works.

Example

classRoot{

str:string;

}

classChildextendsRoot{}

classLeafextendsChild{}//indirectly inherits from Root by virtue of inheritance

var obj =newLeaf();

obj.str ="hello"

console.log(obj.str)

The class Leaf derives the attributes from Root and Child classes by virtue of multi-level inheritance.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var \_\_extends =(this&&this.\_\_extends)||function(d, b){

for(var p in b)if(b.hasOwnProperty(p)) d[p]= b[p];

function \_\_(){this.constructor = d;}

d.prototype = b ===null?Object.create(b):(\_\_.prototype = b.prototype,new \_\_());

};

varRoot=(function(){

functionRoot(){

}

returnRoot;

}());

varChild=(function(\_super){

\_\_extends(Child, \_super);

functionChild(){

\_super.apply(this, arguments);

}

returnChild;

}(Root));

varLeaf=(function(\_super){

\_\_extends(Leaf, \_super);

functionLeaf(){

\_super.apply(this, arguments);

}

returnLeaf;

}(Child));

var obj =newLeaf();

obj.str ="hello";

console.log(obj.str);

Its output is as follows −

Output

hello

TypeScript ─ Class inheritance and Method Overriding

Method Overriding is a mechanism by which the child class redefines the superclass’s method. The following example illustrates the same −

classPrinterClass{

doPrint():void{

console.log("doPrint() from Parent called…")

}

}

classStringPrinterextendsPrinterClass{

doPrint():void{

super.doPrint()

console.log("doPrint() is printing a string…")

}

}

var obj =newStringPrinter()

obj.doPrint()

The super keyword is used to refer to the immediate parent of a class. The keyword can be used to refer to the super class version of a variable, property or method. Line 13 invokes the super class version of the doWork() function.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var \_\_extends =(this&&this.\_\_extends)||function(d, b){

for(var p in b)if(b.hasOwnProperty(p)) d[p]= b[p];

function \_\_(){this.constructor = d;}

d.prototype = b ===null?Object.create(b):(\_\_.prototype = b.prototype,new \_\_());

};

varPrinterClass=(function(){

functionPrinterClass(){

}

PrinterClass.prototype.doPrint=function(){

console.log("doPrint() from Parent called…");

};

returnPrinterClass;

}());

varStringPrinter=(function(\_super){

\_\_extends(StringPrinter, \_super);

functionStringPrinter(){

\_super.apply(this, arguments);

}

StringPrinter.prototype.doPrint=function(){

\_super.prototype.doPrint.call(this);

console.log("doPrint() is printing a string…");

};

returnStringPrinter;

}(PrinterClass));

var obj =newStringPrinter();

obj.doPrint();

The output of the above code is as follows −

doPrint() from Parent called…

doPrint() is printing a string…

The static Keyword

The static keyword can be applied to the data members of a class. A static variable retains its values till the program finishes execution. Static members are referenced by the class name.

Example

classStaticMem{

staticnum:number;

staticdisp():void{

console.log("The value of num is"+StaticMem.num)

}

}

StaticMem.num =12// initialize the static variable

StaticMem.disp()// invoke the static method

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

varStaticMem=(function(){

functionStaticMem(){

}

StaticMem.disp =function(){

console.log("The value of num is"+StaticMem.num);

};

returnStaticMem;

}());

StaticMem.num =12;// initialize the static variable

StaticMem.disp();// invoke the static method

The output of the above code is as follows −

The value of num is 12

The instanceof operator

The **instanceof** operator returns true if the object belongs to the specified type.

Example

classPerson{}

var obj =newPerson()

var isPerson = obj instanceofPerson;

console.log(" obj is an instance of Person "+ isPerson);

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

varPerson=(function(){

functionPerson(){

}

returnPerson;

}());

var obj =newPerson();

var isPerson = obj instanceofPerson;

console.log(" obj is an instance of Person "+ isPerson);

The output of the above code is as follows −

obj is an instance of Person True

Data Hiding

A class can control the visibility of its data members to members of other classes. This capability is termed as Data Hiding or Encapsulation.

Object Orientation uses the concept of access modifiers or access specifiers to implement the concept of Encapsulation. The access specifiers/modifiers define the visibility of a class’s data members outside its defining class.

The access modifiers supported by TypeScript are −

|  |  |
| --- | --- |
| **S.No.** | **Access Specifier & Description** |
| 1. | **public**  A public data member has universal accessibility. Data members in a class are public by default. |
| 2. | **private**  Private data members are accessible only within the class that defines these members. If an external class member tries to access a private member, the compiler throws an error. |
| 3. | **protected**  A protected data member is accessible by the members within the same class as that of the former and also by the members of the child classes. |

Example

Let us now take an example to see how data hiding works −

classEncapsulate{

str:string="hello"

private str2:string="world"

}

var obj =newEncapsulate()

console.log(obj.str)//accessible

console.log(obj.str2)//compilation Error as str2 is private

The class has two string attributes, str1 and str2, which are public and private members respectively. The class is instantiated. The example returns a compile time error, as the private attribute str2 is accessed outside the class that declares it.

Classes and Interfaces

Classes can also implement interfaces.

interfaceILoan{

interest:number

}

classAgriLoanimplementsILoan{

interest:number

rebate:number

constructor(interest:number,rebate:number){

this.interest= interest

this.rebate= rebate

}

}

var obj =newAgriLoan(10,1)

console.log("Interest is : "+obj.interest+" Rebate is : "+obj.rebate )

The class AgriLoan implements the interface Loan. Hence, it is now binding on the class to include the property **interest** as its member.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

varAgriLoan=(function(){

functionAgriLoan(interest, rebate){

this.interest= interest;

this.rebate= rebate;

}

returnAgriLoan;

}());

var obj =newAgriLoan(10,1);

console.log("Interest is : "+ obj.interest +" Rebate is : "+ obj.rebate);

The output of the above code is as follows −

Interest is : 10 Rebate is : 1

TypeScript - Objects

An **object** is an instance which contains set of key value pairs. The values can be scalar values or functions or even array of other objects. The syntax is given below −

Syntax

var object\_name = {

key1: “value1”, //scalar value

key2: “value”,

key3: function() {

//functions

},

key4:[“content1”, “content2”] //collection

};

As shown above, an object can contain scalar values, functions and structures like arrays and tuples.

Example: Object Literal Notation

var person ={

firstname:"Tom",

lastname:"Hanks"

};

//access the object values

console.log(person.firstname)

console.log(person.lastname)

On compiling, it will generate the same code in JavaScript.

The output of the above code is as follows −

Tom

Hanks

TypeScript Type Template

Let’s say you created an object literal in JavaScript as −

var person ={

firstname:"Tom",

lastname:"Hanks"

};

In case you want to add some value to an object, JavaScript allows you to make the necessary modification. Suppose we need to add a function to the person object later this is the way you can do this.

person.sayHello=function(){return"hello";}

If you use the same code in Typescript the compiler gives an error. This is because in Typescript, concrete objects should have a type template. Objects in Typescript must be an instance of a particular type.

You can solve this by using a method template in declaration.

Example: Typescript Type template

var person ={

firstName:"Tom",

lastName:"Hanks",

sayHello:function(){}//Type template

}

person.sayHello=function(){

console.log("hello "+person.firstName)

}

person.sayHello()

On compiling, it will generate the same code in JavaScript.

The output of the above code is as follows −

hello Tom

Objects can also be passed as parameters to function.

Example: Objects as function parameters

var person ={

firstname:"Tom",

lastname:"Hanks"

};

var invokeperson =function(obj:{ firstname:string, lastname :string}){

console.log("first name :"+obj.firstname)

console.log("last name :"+obj.lastname)

}

invokeperson(person)

The example declares an object literal. The function expression is invoked passing person object.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var person ={

firstname:"Tom",

lastname:"Hanks"

};

var invokeperson =function(obj){

console.log("first name :"+ obj.firstname);

console.log("last name :"+ obj.lastname);

};

invokeperson(person);

Its output is as follows −

first name :Tom

last name :Hanks

You can create and pass an anonymous object on the fly.

Example: Anonymous Object

var invokeperson =function(obj:{ firstname:string, lastname :string}){

console.log("first name :"+obj.firstname)

console.log("last name :"+obj.lastname)

}

invokeperson({firstname:"Sachin",lastname:"Tendulkar"});

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var invokeperson =function(obj){

console.log("first name :"+ obj.firstname);

console.log("last name :"+ obj.lastname);

};

invokeperson({ firstname:"Sachin", lastname:"Tendulkar"});

invokeperson({ firstname:"Sachin", lastname:"Tendulkar"});

Its output is as follows −

first name :Sachin

last name :Tendulkar

Duck-typing

In duck-typing, two objects are considered to be of the same type if both share the same set of properties. Duck-typing verifies the presence of certain properties in the objects, rather than their actual type, to check their suitability. The concept is generally explained by the following phrase −

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

The TypeScript compiler implements the duck-typing system that allows object creation on the fly while keeping type safety. The following example shows how we can pass objects that don’t explicitly implement an interface but contain all of the required members to a function.

Example

interfaceIPoint{

x:number

y:number

}

function addPoints(p1:IPoint,p2:IPoint):IPoint{

var x = p1.x + p2.x

var y = p1.y+ p2.y

return{x:x,y:y}

}

//Valid

var newPoint = addPoints({x:3,y:4},{x:5,y:1})

//Error

var newPoint2 = addPoints({x:1},{x:4,y:3})

TypeScript - Namespaces

A namespace is a way to logically group related code. This is inbuilt into TypeScript unlike in JavaScript where variables declarations go into a global scope and if multiple JavaScript files are used within same project there will be possibility of overwriting or misconstruing the same variables, which will lead to the “global namespace pollution problem” in JavaScript.

Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows −

namespaceSomeNameSpaceName{

exportinterfaceISomeInterfaceName{}

exportclassSomeClassName{}

}

The classes or interfaces which should be accessed outside the namespace should be marked with keyword **export**.

To access the class or interface in another namespace, the syntax will be namespaceName.className

SomeNameSpaceName.SomeClassName;

If the first namespace is in separate TypeScript file, then it should be referenced using triple slash reference syntax.

/// <reference path = "SomeFileName.ts" />

The following program demonstrates use of namespaces −

FileName:IShape.ts

----------

namespaceDrawing{

exportinterfaceIShape{

draw();

}

}

FileName:Circle.ts

----------

/// <reference path = "IShape.ts" />

namespaceDrawing{

exportclassCircleimplementsIShape{

publicdraw(){

console.log("Circle is drawn");

}

FileName:Triangle.ts

----------

/// <reference path = "IShape.ts" />

namespaceDrawing{

exportclassTriangleimplementsIShape{

publicdraw(){

console.log("Triangle is drawn");

}

}

FileName:TestShape.ts

/// <reference path = "IShape.ts" />

/// <reference path = "Circle.ts" />

/// <reference path = "Triangle.ts" />

function drawAllShapes(shape:Drawing.IShape){

shape.draw();

}

drawAllShapes(newDrawing.Circle());

drawAllShapes(newDrawing.Triangle());

}

}

}

The above code can be compiled and executed using the following command −

tsc --out app.js TestShape.ts

node app.js

On compiling, it will generate following JavaScript code(app.js).

//Generated by typescript 1.8.10

/// <reference path = "IShape.ts" />

varDrawing;

(function(Drawing){

varCircle=(function(){

functionCircle(){

}

Circle.prototype.draw=function(){

console.log("Cirlce is drawn");

};

returnCircle;

}());

Drawing.Circle=Circle;

})(Drawing||(Drawing={}));

/// <reference path = "IShape.ts" />

varDrawing;

(function(Drawing){

varTriangle=(function(){

functionTriangle(){

}

Triangle.prototype.draw=function(){

console.log("Triangle is drawn");

};

returnTriangle;

}());

Drawing.Triangle=Triangle;

})(Drawing||(Drawing={}));

/// <reference path = "IShape.ts" />

/// <reference path = "Circle.ts" />

/// <reference path = "Triangle.ts" />

function drawAllShapes(shape){

shape.draw();

}

drawAllShapes(newDrawing.Circle());

drawAllShapes(newDrawing.Triangle());

When the above code is compiled and executed, it produces the following result −

Circle is drawn

Triangle is drawn

Nested Namespaces

You can define one namespace inside another namespace as follows −

namespace namespace\_name1 {

exportnamespace namespace\_name2 {

exportclass class\_name {}

}

}

You can access members of nested namespace by using the dot (.) operator as follows −

FileName:Invoice.ts

namespace tutorialPoint {

exportnamespace invoiceApp {

exportclassInvoice{

publiccalculateDiscount(price: number){

return price \*.40;

}

}

}

}

FileName:InvoiceTest.ts

/// <reference path = "Invoice.ts" />

var invoice =newtutorialPoint.invoiceApp.Invoice();

console.log(invoice.calculateDiscount(500));

The above code can be compiled and executed using the following command −

tsc --out app.js InvoiceTest.ts

node app.js

On compiling, it will generate following JavaScript code(app.js).

//Generated by typescript 1.8.10

var tutorialPoint;

(function(tutorialPoint){

var invoiceApp;

(function(invoiceApp){

varInvoice=(function(){

functionInvoice(){

}

Invoice.prototype.calculateDiscount=function(price){

return price \*.40;

};

returnInvoice;

}());

invoiceApp.Invoice=Invoice;

})(invoiceApp = tutorialPoint.invoiceApp ||(tutorialPoint.invoiceApp ={}));

})(tutorialPoint ||(tutorialPoint ={}));

/// <reference path = "Invoice.ts" />

var invoice =newtutorialPoint.invoiceApp.Invoice();

console.log(invoice.calculateDiscount(500));

When the above code is compiled and executed, it produces the following result −

200

TypeScript - Modules

A module is designed with the idea to organize code written in TypeScript. Modules are broadly divided into −

* Internal Modules
* External Modules

Internal Module

Internal modules came in earlier version of Typescript. This was used to logically group classes, interfaces, functions into one unit and can be exported in another module. This logical grouping is named namespace in latest version of TypeScript. So internal modules are obsolete instead we can use namespace. Internal modules are still supported, but its recommended to use namespace over internal modules.

Internal Module Syntax (Old)

module TutorialPoint {

export function add(x, y) {

console.log(x+y);

}

}

Namespace Syntax (New)

namespace TutorialPoint {

export function add(x, y) { console.log(x + y);}

}

JavaScript generated in both cases are same

varTutorialPoint;

(function(TutorialPoint){

functionadd(x, y){

console.log(x + y);

}

TutorialPoint.add = add;

})(TutorialPoint||(TutorialPoint={}));

External Module

External modules in TypeScript exists to specify and load dependencies between multiple external **js** files. If there is only one **js** file used, then external modules are not relevant. Traditionally dependency management between JavaScript files was done using browser script tags (<script></script>). But that’s not extendable, as its very linear while loading modules. That means instead of loading files one after other there is no asynchronous option to load modules. When you are programming js for the server for example NodeJs you don’t even have script tags.

There are two scenarios for loading dependents **js** files from a single main JavaScript file.

* Client Side - RequireJs
* Server Side - NodeJs

Selecting a Module Loader

To support loading external JavaScript files, we need a module loader. This will be another **js** library. For browser the most common library used is RequieJS. This is an implementation of AMD (Asynchronous Module Definition) specification. Instead of loading files one after the other, AMD can load them all separately, even when they are dependent on each other.

Defining External Module

When defining external module in TypeScript targeting CommonJS or AMD, each file is considered as a module. So it’s optional to use internal module with in external module.

If you are migrating TypeScript from AMD to CommonJs module systems, then there is no additional work needed. The only thing you need to change is just the compiler flag Unlike in JavaScript there is an overhead in migrating from CommonJs to AMD or vice versa.

The syntax for declaring an external module is using keyword ‘export’ and ‘import’.

Syntax

//FileName : SomeInterface.ts

export interface SomeInterface {

//code declarations

}

To use the declared module in another file, an import keyword is used as given below. The file name is only specified no extension used.

import someInterfaceRef = require(“./SomeInterface”);

Example

Let’s understand this using an example.

// IShape.ts

exportinterfaceIShape{

draw();

}

// Circle.ts

import shape =require("./IShape");

exportclassCircleimplementsshape.IShape{

publicdraw(){

console.log("Cirlce is drawn (external module)");

}

}

// Triangle.ts

import shape =require("./IShape");

exportclassTriangleimplementsshape.IShape{

publicdraw(){

console.log("Triangle is drawn (external module)");

}

}

// TestShape.ts

import shape =require("./IShape");

import circle =require("./Circle");

import triangle =require("./Triangle");

functiondrawAllShapes(shapeToDraw: shape.IShape){

shapeToDraw.draw();

}

drawAllShapes(new circle.Circle());

drawAllShapes(new triangle.Triangle());

The command to compile the main TypeScript file for AMD systems is −

tsc --module amd TestShape.ts

On compiling, it will generate following JavaScript code for AMD.

File:IShape.js

//Generated by typescript 1.8.10

define(["require","exports"],function(require, exports){

});

File:Circle.js

//Generated by typescript 1.8.10

define(["require","exports"],function(require, exports){

varCircle=(function(){

functionCircle(){

}

Circle.prototype.draw=function(){

console.log("Cirlce is drawn (external module)");

};

returnCircle;

})();

exports.Circle=Circle;

});

File:Triangle.js

//Generated by typescript 1.8.10

define(["require","exports"],function(require, exports){

varTriangle=(function(){

functionTriangle(){

}

Triangle.prototype.draw=function(){

console.log("Triangle is drawn (external module)");

};

returnTriangle;

})();

exports.Triangle=Triangle;

});

File:TestShape.js

//Generated by typescript 1.8.10

define(["require","exports","./Circle","./Triangle"],

function(require, exports, circle, triangle){

function drawAllShapes(shapeToDraw){

shapeToDraw.draw();

}

drawAllShapes(new circle.Circle());

drawAllShapes(new triangle.Triangle());

});

The command to compile the main TypeScript file for **Commonjs** systems is

tsc --module commonjs TestShape.ts

On compiling, it will generate following JavaScript code for **Commonjs**.

File:Circle.js

//Generated by typescript 1.8.10

varCircle=(function(){

functionCircle(){

}

Circle.prototype.draw=function(){

console.log("Cirlce is drawn");

};

returnCircle;

})();

exports.Circle=Circle;

File:Triangle.js

//Generated by typescript 1.8.10

varTriangle=(function(){

functionTriangle(){

}

Triangle.prototype.draw=function(){

console.log("Triangle is drawn (external module)");

};

returnTriangle;

})();

exports.Triangle=Triangle;

File:TestShape.js

//Generated by typescript 1.8.10

var circle =require("./Circle");

var triangle =require("./Triangle");

function drawAllShapes(shapeToDraw){

shapeToDraw.draw();

}

drawAllShapes(new circle.Circle());

drawAllShapes(new triangle.Triangle());

Output

Cirlce is drawn (external module)

Triangle is drawn (external module)

TypeScript - Ambients

Ambient declarations are a way of telling the TypeScript compiler that the actual source code exists elsewhere. When you are consuming a bunch of third party **js** libraries like jquery/angularjs/nodejs you can’t rewrite it in TypeScript. Ensuring typesafety and intellisense while using these libraries will be challenging for a TypeScript programmer. Ambient declarations help to seamlessly integrate other **js** libraries into TypeScript.

Defining Ambients

Ambient declarations are by convention kept in a type declaration file with following extension (d.ts)

Sample.d.ts

The above file will not be transcompiled to JavaScript. It will be used for type safety and intellisense.

The syntax for declaring ambient variables or modules will be as following −

Syntax

declare module Module\_Name {

}

The ambient files should be referenced in the client TypeScript file as shown −

/// <reference path = " Sample.d.ts" />

Example

Let’s understand this with help of an example. Assume you been given a third party javascript library which contains code similar to this.

FileName:CalcThirdPartyJsLib.js

varTutorialPoint;

(function(TutorialPoint){

varCalc=(function(){

functionCalc(){

}

Calc.prototype.doSum=function(limit){

var sum =0;

for(var i =0; i <= limit; i++){

Calc.prototype.doSum=function(limit){

var sum =0;

for(var i =0; i <= limit; i++){

sum = sum + i;

return sum;

returnCalc;

TutorialPoint.Calc=Calc;

})(TutorialPoint||(TutorialPoint={}));

var test =newTutorialPoint.Calc();

}

}

}

}

}

As a typescript programmer you will not have time to rewrite this library to typescript. But still you need to use the doSum() method with type safety. What you could do is ambient declaration file. Let us create an ambient declaration file Calc.d.ts

FileName:Calc.d.ts

declare moduleTutorialPoint{

exportclassCalc{

doSum(limit:number): number;

}

}

Ambient files will not contain the implementations, it is just type declarations. Declarations now need to be included in the typescript file as follows.

FileName:CalcTest.ts

/// <reference path = "Calc.d.ts" />

var obj =newTutorialPoint.Calc();

obj.doSum("Hello");// compiler error

console.log(obj.doSum(10));

The following line of code will show a compiler error. This is because in the declaration file we specified the input parameter will be number.

obj.doSum("Hello");

Comment the above line and compile the program using the following syntax −

tsc CalcTest.ts

On compiling, it will generate following JavaScript code(CalcTest.js).

//Generated by typescript 1.8.10

/// <reference path = "Calc.d.ts" />

var obj =newTutorialPoint.Calc();

// obj.doSum("Hello");

console.log(obj.doSum(10));

In order to execute the code, let us add an html page with script tags as given below. Add the compiled CalcTest.js file and the third party library file CalcThirdPartyJsLib.js.

<html>

<bodystyle="font-size:30px;">

<h1>Ambient Test</h1>

<h2>Calc Test</h2>

</body>

<scriptsrc="CalcThirdPartyJsLib.js"></script>

<scriptsrc="CalcTest.js"></script>

</html>