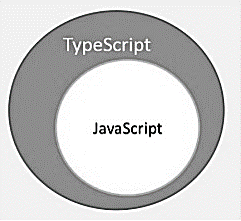
JavaScript was introduced as a language for the client side. The development of Node.js has marked JavaScript as an emerging server-side technology too. However, as JavaScript code grows, it tends to get messier, making it difficult to maintain and reuse the code. Moreover, its failure to embrace the features of Object Orientation, strong type checking and compile-time error checks prevents JavaScript from succeeding at the enterprise level as a full-fledged server-side technology. **Typescript** was presented to bridge this gap.

**What is Typescript?**

Typescript is a strongly typed, object oriented, compiled language. It was designed by **Anders Hejlsberg** (designer of C#) at Microsoft. 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.



**Features of Typescript**

**Typescript is just JavaScript**. Typescript starts with JavaScript and ends with JavaScript. Typescript adopts the basic building blocks of your program from JavaScript. Hence, you only need to know JavaScript to use Typescript. All Typescript code is converted into its JavaScript equivalent for the purpose of execution.

**Typescript supports other JS libraries**. Compiled Typescript can be consumed from any JavaScript code. Typescript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.

**JavaScript is Typescript**. This means that any valid .js file can be renamed to .ts and compiled with other Typescript files.

**Typescript is portable**. Typescript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on. Unlike its counterparts, Typescript doesn’t need a dedicated VM or a specific runtime environment to execute.

**Typescript and ECMAScript**

The ECMAScript specification is a standardized specification of a scripting language. There are six editions of ECMA-262 published. Version 6 of the standard is codenamed "Harmony". Typescript is aligned with the ECMAScript6 specification.



Typescript adopts its basic language features from the ECMAScript5 specification, i.e., the official specification for JavaScript. Typescript language features like Modules and class-based orientation are in line with the EcmaScript 6 specification. Additionally, Typescript also embraces features like generics and type annotations that aren’t a part of the EcmaScript6 specification.

**Why Use Typescript?**

Typescript is superior to its other counterparts like CoffeeScript and Dart programming languages in a way that Typescript is extended JavaScript. In contrast, languages like Dart, CoffeeScript are new languages in themselves and require language-specific execution environment.

The benefits of Typescript include −

**Compilation** − JavaScript is an interpreted language. Hence, it needs to be run to test that it is valid. It means you write all the codes just to find no output, in case there is an error. Hence, you have to spend hours trying to find bugs in the code. The Typescript transpiler provides the error-checking feature. Typescript will compile the code and generate compilation errors, if it finds some sort of syntax errors. This helps to highlight errors before the script is run.

**Strong Static Typing** − JavaScript is not strongly typed. Typescript comes with an optional static typing and type inference system through the TLS (Typescript Language Service). The type of a variable, declared with no type, may be inferred by the TLS based on its value.

Typescript **supports type definitions** for existing JavaScript libraries. Typescript Definition file (with .d.ts extension) provides definition for external JavaScript libraries. Hence, Typescript code can contain these libraries.

Typescript **supports Object Oriented Programming** concepts like classes, interfaces, inheritance, etc.

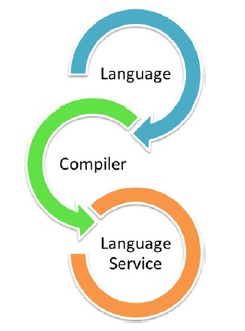
**Components of Typescript**

At its heart, Typescript has the following three components −

**Language** − It comprises of the syntax, keywords, and type annotations.

**The Typescript Compiler** − The Typescript compiler (tsc) converts the instructions written in Typescript to its JavaScript equivalent.

**The Typescript Language Service** − The "Language Service" exposes an additional layer around the core compiler pipeline that are editor-like applications. The language service supports the common set of typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.

  
**Declaration Files**

When a Typescript script gets compiled, there is an option to generate a declaration file (with the extension .d.ts) that functions as an interface to the components in the compiled JavaScript. The concept of declaration files is analogous to the concept of header files found in C/C++. The declaration files (files with .d.ts extension) provide intellisense for types, function calls, and variable support for JavaScript libraries like jQuery, MooTools, etc.

**Environment Setup**

Typescript is an Open Source technology. It can run on any browser, any host, and any OS. You will need the following tools to write and test a Typescript program −

**A Text Editor:** The text editor helps you to write your source code. Examples of a few editors include Windows Notepad, Notepad++, Emacs, vim or vi, etc. Editors used may vary with Operating Systems.

The source files are typically named with the extension .ts

**The Typescript Compiler:** The Typescript compiler is itself a .ts file compiled down to JavaScript (.js) file. 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. In other words, the TSC produces an equivalent JavaScript source code from the Typescript file given as an input to it. This process is termed as transpilation.

However, the compiler rejects any raw JavaScript file passed to it. The compiler deals with only .ts or .d.ts files.

**Installing Node.js**

Node.js is an open source, cross-platform runtime environment for server-side JavaScript. Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute code. You may download Node.js source code or a pre-built installer for your platform. Node is available here − https://nodejs.org/en/download

Follow the steps given below to install Node.js in Windows environment.

**Step 1** − Download and run the .msi installer for Node.

**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***

**IDE Support**

Typescript can be built on a plethora of development environments like Visual Studio, Sublime Text 2, WebStorm/PHPStorm, Eclipse, Brackets, etc.

Syntax defines a set of rules for writing programs. Every language specification defines its own syntax. A TypeScript program is composed of −

* Modules
* Functions
* Variables
* Statements and Expressions
* Comments

**Your First Typescript Code**

Let us start with the traditional “Hello World” example −

var message:string = "Hello World"

console.log(message)

On compiling, it will generate following JavaScript code.

var message = "Hello World";

console.log(message);

**Compile and Execute a Typescript Program**

Let us see how to compile and execute a Typescript program using Visual Studio Code. Follow the steps given below

**Step 1** − Save the file with .ts extension. We shall save the file as Test.ts. The code editor marks errors in the code, if any, while you save it.

**Step 2** − Right-click the Typescript file under the Working Files option in VS Code’s Explore Pane. Select Open in Command Prompt option.

**Step 3** − To compile the file use the following command on the terminal window.

tsc Test.ts

**Step 4** − The file is compiled to Test.js. To run the program written, type the following in the terminal.

node Test.js

**Note** − Multiple files can be compiled at once.

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

**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.

**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

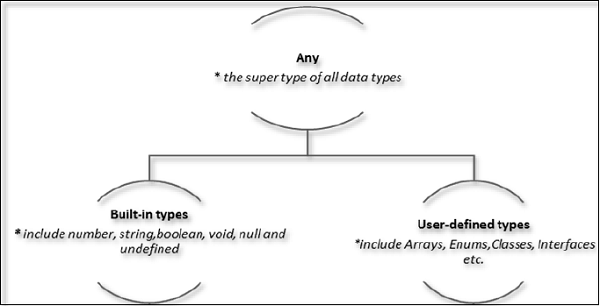
Multi-line comment

\*/

**Typescript Types**

The Type System represents the different types of values supported by the language. The Type System checks the validity of the supplied values, before they are stored or manipulated by the program. This ensures that the code behaves as expected. The Type System further allows for richer code hinting and automated documentation too.

Typescript provides data types as a part of its optional Type System. The data type classification is as given below −



**The Any type**

The any data type is the super type of all types in Typescript. It denotes a dynamic type. Using the any type is equivalent to opting out of type checking for a variable.

**Built-in types**

The following table illustrates all the built-in types in Typescript −

|  |  |  |
| --- | --- | --- |
| 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 |

**Note** − There is no integer type in Typescript and JavaScript.

**Null and undefined ─ Are they the same?**

The null and the undefined data types are often a source of confusion. The null and undefined cannot be used to reference the data type of a variable. They can only be assigned as values to a variable.

However, null and undefined are not the same. A variable initialized with undefined means that the variable has no value or object assigned to it while null means that the variable has been set to an object whose value is undefined.

**User-defined Types**

User-defined types include Enumerations (enums), classes, interfaces, arrays, and tuple.

**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_type.jpg

Declare its type but no value. In this case, the variable will be set to undefined.

declare_type-1.jpg

Declare its value but no type. The variable type will be set to the data type of the assigned value.

declare_type-2.jpg

Declare neither value not type. In this case, the data type of the variable will be any and will be initialized to undefined.

declare_type-3.jpg

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.ts

**Type Assertion in Typescript**

Typescript allows changing a variable from one type to another. Typescript refers to this process as Type Assertion. The syntax is to put the target type between < > symbols and place it in front of the variable or expression. The following example explains this concept −

**Example**

var str = '1'

var str2:number = <number> <any> str //str is now of type number

console.log(typeof(str2))

If you hover the mouse pointer over the type assertion statement in Visual Studio Code, it displays the change in the variable’s data type. Basically it allows the assertion from type S to T succeed if either S is a subtype of T or T is a subtype of S.

The reason why it's not called "type casting" is that casting generally implies some sort of runtime support while, “type assertions” are purely a compile time construct and a way for you to provide hints to the compiler on how you want your code to be analyzed.

On compiling, it will generate following JavaScript code.

"use strict";

var str = '1';

var str2 = str; //str is now of type number

console.log(typeof (str2));

It will produce the following output −

string

**Inferred Typing in Typescript**

Given the fact that, Typescript is strongly typed, this feature is optional. Typescript also encourages dynamic typing of variables. This means that, Typescript encourages declaring a variable without a type. In such cases, the compiler will determine the type of the variable on the basis of the value assigned to it. Typescript will find the first usage of the variable within the code, determine the type to which it has been initially set and then assume the same type for this variable in the rest of your code block.

The same is explained in the following code snippet −

**Example**: Inferred Typing

var num = 2; // data type inferred as number

console.log("value of num "+num);

num = "12";

console.log(num);

It will produce the following output −

error TS2011: Cannot convert 'string' to 'number'.

**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.

The following example illustrates variable scopes in TypeScript.

**Example**: variable\_scope.ts

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

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

**What is an Operator?**

An operator defines some function that will be performed on the data. The data on which operators work are called operands. Consider the following expression −

**7 + 5 = 12**

Here, the values 7, 5, and 12 are operands, while + and = are operators.

The major operators in TypeScript can be classified as −

* Arithmetic operators
* Logical operators
* Relational operators
* Bitwise operators
* Assignment operators
* Ternary/conditional operator
* String operator
* Type Operator

**Arithmetic Operators**

Assume the values in variables a and b are 10 and 5 respectively.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| + (Addition) | returns the sum of the operands | a + b is 15 |
| - (Subtraction) | returns the difference of the values | a - b is 5 |
| \* (Multiplication) | returns the product of the values | a \* b is 50 |
| / (Division) | performs division operation and returns the quotient | a / b is 2 |
| % (Modulus) | performs division operation and returns the remainder | a % b is 0 |
| ++ (Increment) | Increments the value of the variable by one | a++ is 11 |
| -- (Decrement) | Decrements the value of the variable by one | a-- is 9 |

**Example**: arithmetic.ts

**Relational Operators**

Relational Operators test or define the kind of relationship between two entities. Relational operators return a Boolean value, i.e., true/ false.

Assume the value of A is 10 and B is 20.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| > | Greater than | (A > B) is False |
| < | Lesser than | (A < B) is True |
| >= | Greater than or equal to | (A >= B) is False |
| <= | Lesser than or equal to | (A <= B) is True |
| == | Equality | (A == B) is false |
| != | Not equal | (A != B) is True |

**Example**: relational.ts

**Logical Operators**

Logical Operators are used to combine two or more conditions. Logical operators too return a Boolean value. Assume the value of variable A is 10 and B is 20.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| && (And) | The operator returns true only if all the expressions specified return true | (A > 10 && B > 10) is False |
| || (OR) | The operator returns true if at least one of the expressions specified return true | (A > 10 || B >10) is True |
| ! (NOT) | The operator returns the inverse of the expression’s result. For E.g.: !(>5) returns false | !(A >10 ) is True |

**Example**: logical.ts

**Bitwise Operators**

Assume variable A = 2 and B = 3

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| & (Bitwise AND) | It performs a Boolean AND operation on each bit of its integer arguments. | (A & B) is 2 |
| | (BitWise OR) | It performs a Boolean OR operation on each bit of its integer arguments. | (A | B) is 3 |
| ^ (Bitwise XOR) | It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. | (A ^ B) is 1 |
| ~ (Bitwise Not) | It is a unary operator and operates by reversing all the bits in the operand. | (~B) is -4 |
| << (Left Shift) | It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying it by 2, shifting two positions is equivalent to multiplying by 4, and so on. | (A << 1) is 4 |
| >> (Right Shift) | Binary Right Shift Operator. The left operand’s value is moved right by the number of bits specified by the right operand. | (A >> 1) is 1 |
| >>> (Right shift with Zero) | This operator is just like the >> operator, except that the bits shifted in on the left are always zero. | (A >>> 1) is 1 |

**Example**: bitwise.ts

**Assignment Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| = (Simple Assignment) | Assigns values from the right side operand to the left side operand | C = A + B will assign the value of A + B into C |
| += (Add and Assignment) | It adds the right operand to the left operand and assigns the result to the left operand. | C += A is equivalent to C = C + A |
| -= (Subtract and Assignment) | It subtracts the right operand from the left operand and assigns the result to the left operand. | C -= A is equivalent to C = C – A |
| \*= (Multiply and Assignment) | It multiplies the right operand with the left operand and assigns the result to the left operand. | C \*= A is equivalent to C = C \* A |
| /= (Divide and Assignment) | It divides the left operand with the right operand and assigns the result to the left operand. | C /= A is equivalent to C = C / A |

**Note** − same logic applies to Bitwise operators, so they will become <<=, >>=, >>=, &=, |= and ^=.

**Example**: assignment.ts

**Miscellaneous Operators**

**The negation operator (-)**

Changes the sign of a value. Let’s take an example.

var x:number = 4

var y = -x;

console.log("value of x: ",x); //outputs 4

console.log("value of y: ",y); //outputs -4

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var x = 4;

var y = -x;

console.log("value of x: ", x); //outputs 4

console.log("value of y: ", y); //outputs -4

It will produce the following output −

value of x: 4

value of y: -4

**String Operators: Concatenation operator (+)**

The + operator when applied to strings appends the second string to the first. The following example helps us to understand this concept.

var msg:string = "hello"+"world"

console.log(msg)

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var msg = "hello" + "world";

console.log(msg);

It will produce the following output −

helloworld

The concatenation operation doesn’t add a space between strings. Multiple strings can be concatenated in a single statement.

**Conditional Operator (?)**

This operator is used to represent a conditional expression. The conditional operator is also sometimes referred to as the ternary operator. The syntax is as given below −

Test ? expr1 : expr2

Test − refers to the conditional expression

expr1 − value returned if the condition is true

expr2 − value returned if the condition is false

Let’s take a look at the following code −

var num:number = -2

var result = num > 0 ?"positive":"non-positive"

console.log(result)

Line 2 checks whether the value in the variable num is greater than zero. If num is set to a value greater than zero, it returns the string “positive” else the string “non-positive” is returned.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var num = -2;

var result = num > 0 ? "positive" : "non-positive";

console.log(result);

The above code snippet will produce the following output −

non-positive

**Type Operators**

**typeof operator**

It is a unary operator. This operator returns the data type of the operand. Take a look at the following example −

var num = 12

console.log(typeof num); //output: number

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var num = 12;

console.log(typeof num); //output: number

It will produce the following output −

number

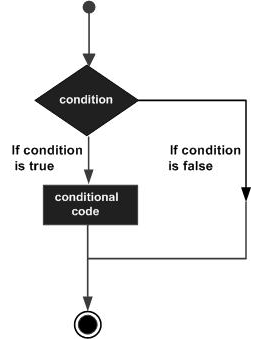
**instanceof**

This operator can be used to test if an object is of a specified type or not. The use of instanceof operator is discussed in the chapter classes.

**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  An ‘if’ statement consists of a Boolean expression followed by one or more statements. |
| 2. | if...else statement  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  You can use one ‘if’ or ‘else if’ statement inside another ‘if’ or ‘else if’ statement(s). |
| 4. | switch statement  A ‘switch’ statement allows a variable to be tested against a list of values. |

**Example**: if.ts

**Example**: ifelse.ts

**Example**: ifelseif.ts

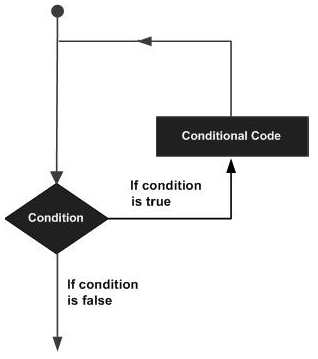
**Example**: switch.ts

**Typescript – Loops**

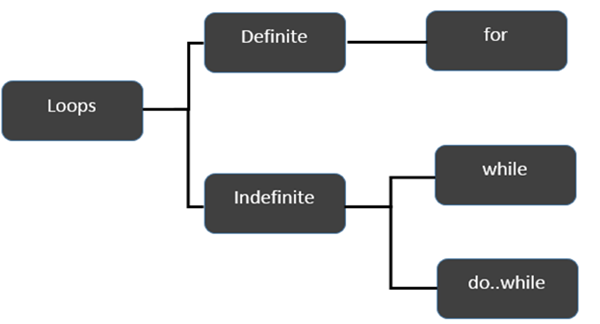
You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

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 numbers 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  The for loop is an implementation of a definite loop. |

**Example**: forloop.ts

**Example**: forin.ts

**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  The while loop executes the instructions each time the condition specified evaluates to true. |
| 2. | do… while  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**: whileloop.ts

**Example**: dowhile.ts

**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.

**Example**

var i: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.

**Example**

An example of the continue statement is given below −

var num:number = 0

var count: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(“This is an endless loop”)

}

Syntax: Infinite loop using while loop

while(true) {

//statements

}

**Example**: Infinite loop using while loop

while(true) {

console.log(“This is 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.

**Example**: Defining and calling a function [function1.ts]

**Example**: Function that returns value [function2.ts]

**Example**: Positional parameters [function3.ts]

**Example**: Optional Parameters [function4.ts]

**NOTE**: The optional parameter should be set as the last argument in a function.

**Example**: Rest Parameters (Variable Arguments) [function5.ts]

**Example**: Default parameters [function6.ts]

**Example** ─ A Simple Anonymous function [function7.ts]

**Example** ─ Anonymous function with parameters [function8.ts]

**The Function Constructor**

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

Syntax

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

**Example** function9.ts

**Example** – Recursion [function10.ts]

**Example**: Anonymous Recursive Function [function11.ts]

**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 [function12.ts]

**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**: function13.ts

**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.

**Example**: function14.ts

**Example**: Optional parentheses for a single parameter [function15.ts]

**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.

**Example**: function16.ts

**Typescript - Arrays**

* 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**

* 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**

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

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

Arrays may be declared and initialized in a single statement.

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

**Example**: array1.ts

**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.

**Example:** array2.ts

**Array Methods**

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

**Example**: array\_methods.ts

**Array Destructuring**

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

**Example**: array4.ts

**Array Traversal using for…in loop**

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

**Example**: array5.ts

**Arrays in Typescript**

|  |  |  |
| --- | --- | --- |
| S.No | Concept and Description | Example |
| 1 | **Multi-dimensional arrays**: Typescript supports multidimensional arrays. The simplest form of the multidimensional array is the two dimensional array. | array6.ts |
| 2 | **Passing arrays to functions:** You can pass to the function a pointer to an array by specifying the array's name without an index. | array7.ts |
| 3 | **Return array from functions:** Allows a function to return an array | array8.ts |

**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.ts

**Example**: empty\_tuple.ts

**Tuple Operations**

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

**Example**: tuple\_ops.ts

**Updating Tuples**

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

**Example**: tuple\_update.ts

**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**: tuple\_destruct.ts

**Typescript - Union**

Typescript 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**: union1.ts

**Example**: union2.ts (Union Type and function parameter)

**Union Type and Arrays**

Union types can also be applied to arrays, properties and interfaces.

**Example**: union3.ts (Example: Union Type and Array)

**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**: interface1.ts (Interface and Objects)

Interfaces are not to be converted to JavaScript. It’s just part of Typescript. So interfaces have zero runtime JavaScript impact.

**Example**: interface2.ts (Union Type and Interface)

**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**: interface3.ts (Simple Interface Inheritance)

**Example**: interface4.ts(Multiple Interface Inheritance)

**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.

class Person {

}

**Example: Declaring a class**

class Car {

//field

engine:string;

//constructor

constructor(engine:string) {

this.engine = engine

}

//function

disp():void {

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

}

}

**Creating Instance objects**

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

Syntax

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

The new keyword is responsible for instantiation.

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

Example: Instantiating a class

var obj = new Car("Engine 1")

**Accessing Attributes and Functions**

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

//accessing an attribute

obj.field\_name

//accessing a function

obj.function\_name()

**Example**: class1.ts

**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 private members and constructors from the parent class.

Syntax

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

However, TypeScript doesn’t support multiple inheritance.

**Example**: class2.ts

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**: class3.ts

**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 −

**Example**: class4.ts

**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**: class5.ts

**The instanceof operator**

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

**Example**: class6.ts

**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**: class7.ts (Classes and Interfaces)

**Typescript - object**

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**: object1.ts (Object Literal Notation)

**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**: object2.ts (Typescript Type template)

**Example**: object3.ts (Objects as function parameters)

**Example**: object4.ts (Anonymous Object)

**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 −

namespace SomeNameSpaceName {

export interface ISomeInterfaceName { }

export class SomeClassName { }

}

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" />

**Example1**: IShape.ts, Circle.ts, Triangle.ts, TestShape.ts

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

tsc --out app.js TestShape.ts

node app.js

**Nested Namespaces**

You can define one namespace inside another namespace as follows −

namespace namespace\_name1 {

export namespace namespace\_name2 {

export class class\_name { }

}

}

**Example2**: Invoice.ts, and InvoiceTest.ts

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

tsc --out app.js InvoiceTest.ts

node app.js

**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 MyModule {

export function add(x, y) {

console.log(x+y);

}

}

**Namespace Syntax (New)**

namespace MyNamespace {

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

}

**External Modules**

External modules in Typescript exists to specify and load dependencies between multiple external js files.

**Defining External Module**

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**: IShape.ts, Circle.ts, Triangle.ts, and TestShape.ts

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

tsc --module amd TestShape.ts

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

tsc --module commonjs TestShape.ts

To execute

node TestShape.js

**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" />