# TypeScript Tutorial for Beginners - Quick Start

TypeScript is open- source, has powerful type-checking abilities, and amongst many other features, provides auto-completion, navigation and refactoring, which are very useful in large projects.

**Abstract:**A TypeScript Quick Start tutorial for beginners to get you started with Constructs, Classes, Interfaces, Functions, Decorators, Modules, Transpilation and much more.

This tutorial talks about TypeScript, a typed superset of JavaScript that compiles to plain JavaScript. To try out the examples shown in this article, you can use any open source web editor that supports TypeScript like the [**Visual Studio Code**](http://bit.ly/dncvscode) or an online playground like <https://www.typescriptlang.org/play/>.

This tutorial is based on TypeScript v1.8.

**TypeScript Introduction**

Using the JavaScript language to build massive scale web applications, APIs and even mobile applications, is not an easy task. JavaScript (based on EcmaScript specification)  was initially designed to perform simple operations in browsers. Ever since then, the ECMAScript specification has been revised to add new syntax and features to write complex applications.

The additions to the language in ECMAScript 6 (a.k.a ES2015 or ES6) and the new features of ECMAScript 7 (a.k.a ES2016), which are under development have made the language a better fit for big applications. Though these additions are great, they couldn’t bring all advantages of a typical typed language to JavaScript. JavaScript was never meant to be a typed language and the TC39 committee never wantede to take the language in that direction. In addition to these challenges, browsers also need time to implement these features.

This article is published from the **DNC Magazine for Developers and Architects**. [**Download this magazine from here**](http://www.dotnetcurry.net/s/dnc-mag-24th-zip) [Zip PDF] or [**Subscribe to this magazine for FREE**](http://www.dotnetcurry.com/magazine/) and download all previous and current editions

**Microsoft’s solution to these challenges is TypeScript – a superset of JavaScript**. TypeScript is a open source programming language created and maintained by Microsoft and was first announced in October 2012. Let’s understand what the language is about and how it works.

**What is TypeScript?**

TypeScript is an open source programming language written on top of JavaScript to support types. TypeScript is not a completely new language and is not intended to replace JavaScript in anyway. It adheres to all principles of JavaScript and just adds types on top of it. If you are already good at JavaScript, it won’t take you much time to learn TypeScript.

TypeScript can be used to write everything that can be written in JavaScript. With support of types, it looks very close to any other typed OOPs language like C# and Java. The types are optional, so it is not mandatory to strongly type everything. However good usage of types has many advantages. For example it enhances the productivity of the team. Strict type checking system makes the code more predictable. The type system of the language helps in solving many issues during development which could not be caught until runtime. This feature reduces the development time of a team. As most of the issues are resolved upfront, it reduces the cost of fixing the bugs later.

The TypeScript team is working very hard to keep the language updated with the latest specifications of JavaScript. Most of the proposed features of ES6 and ES7 are implemented by TypeScript. They get converted into their ES5 equivalent when the TypeScript files are *transpiled*. **This makes it possible to use the latest features of JavaScript even when the browsers have not implemented them natively**.

An existing JavaScript library need not be rewritten to be used in TypeScript. We need a type definition file declaring types for the APIs exposed by the library. The GitHub project [Definitely Typed](https://github.com/DefinitelyTyped/DefinitelyTyped) does a great job of creating and maintaining type definition files for most of the JavaScript libraries. These files are made available through a package manager to make it easier to install and use the type definition files.

**TypeScript Tutorial**

**Basic TypeScript constructs: Types, Functions, Classes   
Data Types and Variables**

TypeScript defines a set of basic and general purpose types. They are:

· **number:** All numeric values are treated as numbers. A variable of number type gets access to all APIs of the type *Number*defined by the browser

· **string:** Any valid JavaScript string value gets string type. The string APIs defined by the browser are accessible to this type

· **boolean:** A boolean type variable can be assigned with true or false. Default value of a Boolean variable is false

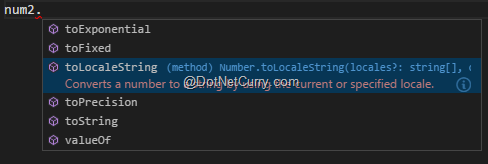
· **any:** A generic type. A variable whose type cannot be represented using any of the basic types, types defined by the browser or the custom classes written in the application, can be declared using the *any* type

· **Empty object type ({}):** This type represents objects that do not have any instance members or members that cannot be added to the object. Hence these are just empty objects. This type is not used in most of the cases.

Syntax of declaring a variable is similar to the way we do it in JavaScript, except we declare type along with it.

|  |
| --- |
| var num: number;  var num2 = 10; |

The variable *num2*infers its type based on the value assigned to it, which is number. When we try to access members of the variable *num2,*auto completion suggests the members defined in the *Number*type. Following is a screenshot from Visual Studio Code (a free code editor from Microsoft) showing auto completion when we access members of *num2*:



Declaring a variable without using a type and without assigning any value to it makes type of the variable *any*. If types of the variable and the value assigned to it differ, the TypeScript compiler generates an error.

**TypeScript Functions**

TypeScript functions can have typed arguments and a return type. TypeScript doesn’t allow to pass variables of different types into the function unless there are explicit declarations of the function with those types.

Consider the following function and the statements that calls the function:

|  |
| --- |
| function add(val1: number, val2: number): number{    return val1 + val2;  }    var numResult = add(10, 20);  var strResult = add("The number is: ", "10"); |

Out of the two statements, the second won’t compile as the function expects the parameters to be numbers and we are passing strings here. Return type of the function is a number, so the variable *numResult*also gets the type number.

It is possible to make the function accept either a number or a string using union types. We need to define a union type to represent number or string and use it as the type in the function definition. Following snippet shows how to rewrite the above function using union types:

|  |
| --- |
| type myType = number | string;    function add(val1: myType, val2: myType): myType{    if((typeof val1 === 'number' && typeof val2 === 'number'))      return val1 + val2;    else if(typeof val1 === 'string' && typeof val2 === 'string')      return val1 + val2;      return null;  }    var result1 = add(10, 20);  var result2 = add("The number is: ", "10");  var result3 = add("The number is:", 10); |

The function adds the parameters when both are either numbers or strings. If both types of both parameters are different, it returns a null. Out of the three calls made to the function, we get result in the first two instances and the third one returns null.

**TypeScript Classes**

Classes are used to define blueprint of objects. They are used extensively in OOPs based languages. Though JavaScript didn’t have direct support for classes till ES6, we were still able to create functionality similar to classes using the *prototype*property of the objects. TypeScript had support for classes since its inception.

Classes in TypeScript can have properties and members as instance variables. Their access to the outside world can be controlled using access specifiers. The class can have a constructor to initialize its members. Arguments of the constructor can be automatically converted to instance members if an access specifier is specified in the argument. Following is an example of a class:

|  |
| --- |
| class Employee{    private bonus: number;      constructor(private empNo: string, private name: string, private salary: number){      this.bonus = this.salary \* 0.1;    }      getDetails(){      return `Employee number is ${this.empNo} and name is ${this.name}`;    }      get Name() {     return this.name;    }      set Name(name: string){     this.name = name;    }  } |

As the arguments passed to the constructor are marked as private, they are made members of the class and they receive the values that are passed to the constructor. The function *getDetails*returns a string containing details of the employee.

The default access specifier in TypeScript is public. Here, the method *getDetails*becomes *public*. It can be called using any object of the class. The field bonus shouldn’t be accessed directly using the object, so it is explicitly made private.

The property *Name*with public getter and setter blocks are encapsulating the private field *name*in the class. This is to prevent direct access to the instance member from outside of the class. And the field *bonus*just has a getter block around it which allows it to be accessed from outside the class and prevent assigning any value to it.

We can instantiate a class using the new keyword. Public members of the class can be accessed using the object created.

|  |
| --- |
| var emp = new Employee("E001", "Alex", 10000);  emp.Name = "Alisa";    console.log(emp.getDetails());  console.log(emp.Bonus); |

**Inheritance in TypeScript**

Reusability is a word that we cannot ignore as programmers. We come across this word every now and then. One of the several ways in which code reusability can be achieved while dealing with classes and objects is through **inheritance**. A class can be inherited from an existing class. The new class gets members of the parent class along with its own.

The child class gets access to all *protected*and *public*members of the parent class and not to the *private*members. When a child class is instantiated, the parent class also gets instantiated. The child class can call constructor of the parent class using the *super*keyword.

Let’s modify the *Employee*class defined above so that some of its members would be accessible through a child class and define a new class, *Manager*to see how inheritance works.

|  |
| --- |
| class Employee{    private bonus: number;      constructor(protected empNo: string, protected name: string, protected salary: number){      this.bonus = this.salary \* 0.1;    }      getDetails(){      return `Employee number is ${this.empNo} and name is ${this.name}`;    }      get Name() {     return this.name;    }      set Name(name: string){     this.name = name;    }      get Bonus(){      return this.bonus;    }  }    class Manager extends Employee {    constructor(empNo: string, name: string, salary: number, private noOfReportees: number) {      super(empNo, name, salary);    }      getDetails(){      var details = super.getDetails();      return `${details} and has ${this.noOfReportees} reportees.`;    }  } |

In constructor of the *Manager*class, we call the constructor of the *Employee*class in the first statement. The *super*keyword in the child class refers to the current instance of the parent class. Call to the parent class constructor has to be the first statement in the child class. If the parent class has a parameterized constructor, it is mandatory to call the parent class constructor.

Also, observe the way the *getDetails*method calls the parent class method. This method overrides the parent class method. When *getDetails*is invoked using an object of the *Manager*class, it calls method of the *Manager*class. This completely hides the parent class method. But the parent class method is available for the child class to invoke, and it can be done using the *super*keyword.

**Interfaces in TypeScript**

Interfaces are used to create contracts. They don’t provide a concrete meaning to anything, they just declare the methods and fields. Because of this, an interface cannot be used as-is to build anything. An interface is meant to be inherited by a class and the class implementing the interface has to define all members of the interface.

Consider the following interface:

|  |
| --- |
| interface IShape{    area(): number;  } |

The above interface represents a shape and declares a method to calculate surface area of the shape. It doesn’t cover any details about the shape – for e.g. the kind of shape, if it has length and breadth or any other details. These details have to be provided by the implementing class. Let’s define two classes, *Square*and *Rectangle*by implementing this interface.

|  |
| --- |
| class Square implements IShape{    constructor(private length: number){}      get Length(){      return this.length;    }      area(){      return this.length \* this.length;    }  }    class Rectangle implements IShape{    constructor(private length: number, private breadth: number){}      area(): number{      return this.length \* this.breadth;    }  } |

We can instantiate these classes and assign them to references of the interface type. We can access the members declared in the interface using this instance.

|  |
| --- |
| var square: IShape = new Square(10);  var rectangle: IShape = new Rectangle(10, 20);  console.log(square.area());  console.log(rectangle.area()); |

The class *Square*class has an additional property *Length*that is not declared in the interface. Though the object *square*is an instance of *Square*class, we cannot access the members that are defined in the class and not in the interface. However, the object *square*can be casted to the type *Square*and then we can use the members defined in the *Square*class.

|  |
| --- |
| var squareObj = square as Square;  console.log(squareObj.Length); |

**TypeScript Decorators**

Decorators are used to extend the behavior without modifying the implementation. Decorators can be applied on classes, members of classes, functions or even on arguments of function. It is a feature proposed for ES7 and is already in use by some of the JavaScript frameworks including Angular 2.

Creating and using decorators is very easy. A custom decorator is a function that accepts some arguments containing details of the target on which it is applied. It can modify the way the target works using this information.

The following snippet defines and uses a decorator:

|  |
| --- |
| function nonEnumerable(target, name, descriptor){    descriptor.enumerable = false;    return descriptor;  }    class Person {    fullName: string;      @nonEnumerable    get name() { return this.fullName; }      set name(val) {      this.fullName = val;    }      get age(){      return 20;    }  }    var p = new Person();  for(let prop in p){    console.log(prop);  } |

The decorator *nonEnumerable*sets enumerable property of a field to false. After this, the property won’t be encountered when we run a *for…in* loop on the object. The loop written at the end of the snippet prints only the property age .

**TypeScript Modules**

Writing applications consisting of hundreds of files is almost impossible without a modular approach. As TypeScript was created to solve the issues with creating large applications using JavaScript, support for modules was added to it. After announcement of the ES6 module system, the module system of the language was renamed to namespaces and ES6 module system is now a first class citizen in TypeScript.

ES6 module system treats every file as a module. A module can export the objects that it wants to make them available to the other modules and also import the objects exported by other modules. In a typical application using ES6 modules, one module made responsible to perform initial task of the application loads a few other modules of the application. These modules in turn load other modules in the application and so on.

A module can export any number of functions, classes or variables. By default, the objects are exported with their original names. We can change this, if required. A module can have a default exported member as well. Following snippet shows examples of different export statements:

|  |
| --- |
| //Inline export with same name  export class MyClass{}  export function myFunc(){}    //Exporting a group of members  export {    MyClass,    myFunc  };      //Rename while exporting  export {    MyClass as AnotherClass,    myFunc as anotherFunc  };    //Default export  export default myMemberToExport; |

When a module imports another module, it may import all exported members, some of them or none at all. The importing module also has the option to rename the importing object. The following snippet shows examples of different import statements:

|  |
| --- |
| //Importing all exported objects  import \* as module1 from "./module1";    //Importing selected objects  import {MyClass1, MyClass2} from "./module1";    //Importing selected objects and rename them  import {MyClass1 as Module1MyClass1, MyClass2 as Module1MyClass2} from "./module1";    //Importing default export object  import d from "./module1"; |

**Transpiling TypeScript and using on a web page**

TypeScript is not created for the browsers. It is a language to be used by developers to write code. . Hence we need to transpile the TypeScript code to JavaScript to be able to run it on the browser. There are several ways in which TypeScript code can be transpiled to JavaScript, each of them can be used in different scenarios depending upon the need.

**Transpiling TypeScript on the Fly**

The easiest way to try TypeScript is by transpiling it on the fly. This approach works for demos really well, but it is not the recommended way for production. Let’s see how the on-the-fly transpiler works before jumping into pre-compiled script loading. To run TypeScript directly in the browser, we need the following libraries:

* **TypeScript transpiler:** a JavaScript file that takes care of transpiling TypeScript on the browser
* **SystemJS module loader:** SystemJS is a universal module loader. It loads all types of JavaScript modules. It is used to load ES6 modules without transpiling them
* **SystemJS polyfill:** used to polyfill the missing module systems. For example, if SystemJS is used to load an AMD style module and no AMD library is loaded on the page, then it polyfills the AMD module system
* **ES6 Shim:** used to polyfill ES6 promises in unsupported browsers
* The HTML page running TypeScript code has to refer to the above four libraries. Following is the content of an HTML page designed for this purpose:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>    <title>Transpiling TypeScript in the browser</title>    <meta name="viewport" content="width=device-width, initial-scale=1">      <a href="https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.35.0/es6-shim.min.js" rel="nofollow" target="\_blank">https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.35.0/es6-shim.min.js</a>    <a href="https://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.25/system-polyfills.js" rel="nofollow" target="\_blank">https://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.25/system-polyfills.js</a>      <a href="https://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.25/system.js" rel="nofollow" target="\_blank">https://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.25/system.js</a>    <a href="https://npmcdn.com/typescript@1.8.9/lib/typescript.js" rel="nofollow" target="\_blank">https://npmcdn.com/typescript@1.8.9/lib/typescript.js</a>  </head>    <body>    <div>Area of the square is:</div>      <div>Area of the rectangle is:</div>  </body>  </html> |

The TypeScript file has the classes *Square* and *Rectangle*we saw earlier in an example, and both of them calculate area. The file exports these two classes. Following is the content of this file:

|  |
| --- |
| export class Square{    constructor(private length: number){}      area(){      return this.length\*this.length;    }  }    export class Rectangle{    constructor(private length: number, private breadth: number){}      area(): number{      return this.length \* this.breadth;    }  } |

It is saved in a file named app.ts. We need to load this file using System.js in the index.html and use the exported members. Following snippet shows how to load the file and use the exported members:

|  |
| --- |
| System.config({    transpiler: 'typescript',    typescriptOptions: { emitDecoratorMetadata: true },    packages: {'.': {defaultExtension: 'ts'}}  });  System.import('app')  .then(function(result) {    var s = new result.Square(10);    var r = new result.Rectangle(10, 20);    document.querySelector("#areaOfSquare").innerHTML = s.area();    document.querySelector("#areaOfRectangle").innerHTML = r.area();  }); |

This page can’t be loaded in the browser directly. It has to be loaded from a server. To start a server easily, you can use the global npm *http-server*to start a Node.js server and open the URL depending upon the port number where it starts the server. Setting up Node and starting a server is covered at http://www.dotnetcurry.com/nodejs/1203/create-web-server-nodejs.

**Pre-transpiling TypeScript using command line**

TypeScript can be pre-compiled to JavaScript using a number of approaches like using command line interface (CLI) of TypeScript, using task runners such as Grunt or Gulp, or using module loaders like Webpack and JSPM. In this chapter, we will see how to pre-compile using the command line interface. We will discuss about Webpack in a future tutorial.

To use TypeScript’s CLI, we need to install the global npm package for TypeScript using the following command:

|  |
| --- |
| > npm install –g typescript |

After successful completion of this command, we can use the *tsc*command to transpile the TypeScript files.

|  |
| --- |
| > tsc app.ts |

As we are using ES6 style modules, we need to tell TypeScript the target module format we are looking for. TypeScript converts the modules into CommonJS format if no module system is specified. If we need to convert into any module type other than CommonJS, we need to specify it in the command line option. Following command converts the module to SystemJS:

> tsc app.ts --module system

This command produces a file named app.js. We can load this file in the HTML file using the SystemJS module loader. We need to make some minor changes to the SystemJS code we wrote earlier to load this file. Following is the modified code:

|  |
| --- |
| System.config({    packages: {'.': {defaultExtension: 'js'}}  });  System.import('app')  .then(function(result){    var s = new result.Square(10);    var r = new result.Rectangle(10, 20);    document.querySelector("#areaOfSquare").innerHTML = s.area();    document.querySelector("#areaOfRectangle").innerHTML = r.area();  }); |

**Conclusion**

TypeScript makes the experience of working with JavaScript better. The language is thoughtfully designed to not invent anything new and yet make the experience of programmers working on JavaScript better. This **TypeScript tutorial**gave you a quick start to TypeScript which will help you to write better Angular 2 apps.