{Learn TypeScript}

Step by Step Guide

About Developer

Ahmer Ali Ahsan

ahmer.ali.ahsan@outlook.com

TYPE SCRIPT

Filename Extension: ts
Developer: Microsoft

JS (JavaScript) VS TS (TypeScript)

JavaScript applications such as web e-mail, maps, document editing, and collaboration tools are becoming an increasingly important part of the everyday computing. We designed TypeScript to meet the needs of the JavaScript programming teams that build and maintain large JavaScript programs.

What is TypeScript?

TypeScript (TS) is an open source programming language developed by Microsoft. Typescript is a typed superset of JavaScript that compiles to plain JavaScript.

Why we need to use TypeScript?

It offers classes, modules, and interfaces to help you build robust components. Classes enable programmers to express common object-oriented patterns in a standard way, making features like inheritance more readable and interoperable. Modules enable programmers to organize their code into components while avoiding naming conflicts. The TypeScript compiler provides module code generation options that support either static or dynamic loading of module contents.

After your TypeScript is compiled to JavaScript, you can use all your other tools—minifiers, packagers, runtime loaders, unit test frameworks, and so forth—as you would if you had written the JavaScript from scratch.

TypeScript syntax includes all features of ECMAScript 2015, including classes and modules, and provides the ability to translate these features into ECMAScript 3 or 5 compliant code.

About ECMAScript:

ECMA stands for - European Computer Manufacturer's Association. ECMAScript is a standard for a scripting language. It specifies the core features that a scripting language should provide and how those features should be implemented. Javascript was originally created at Netscape, and they wanted to standardize the language. So, they submitted the language to the European Computer Manufacturer's Association (ECMA) for standardization. But, there were trademark issues with the name Javascript, and the standard became called ECMAScript, which is the name it holds today as well.

Code Example in TypeScript & JavaScript:

TS	JS

//Our TypeScript HelloWorld class
 	
class HelloWorld {
 	
  //A variable of type HTMLElement
  element: HTMLElement;
 	
  //A Consctructor that accepts an element
  constructor (e: HTMLElement) {
  this.element = e;
  }
 	
  //A public method
  sayHello(message: string) {
  this.element.innerHTML = message;
  }
 	
}
 	
window.onload = () => {
  var e = document.getElementById('content');
 	
  //Initiate HelloWorld Class
  var hello = new HelloWorld(e);
  hello.sayHello("Hello World");
};
	var HelloWorld = (function () {
	  function HelloWorld(e) {
	  this.element = e;
	  }
	  HelloWorld.prototype.sayHello = function (message) {
	  this.element.innerHTML = message;
	  };
	  return HelloWorld;
	})();
	window.onload = function () {
	  var e = document.getElementById('content');
	  var hello = new HelloWorld(e);
	  hello.sayHello("Hello World");
	};

For brief discussion on TypeScript see below links:

- http://stackoverflow.com/questions/12694530/what-is-typescript-and-why-would-i-use-it-in-place-of-javascript
- http://www.codeproject.com/Articles/730843/What-is-TypeScript
- https://raw.githubusercontent.com/Microsoft/TypeScript/master/doc/TypeScript%20Language%20Specification.pdf
- https://blogs.msdn.microsoft.com/somasegar/2012/10/01/typescript-javascript-development-at-application-scale/

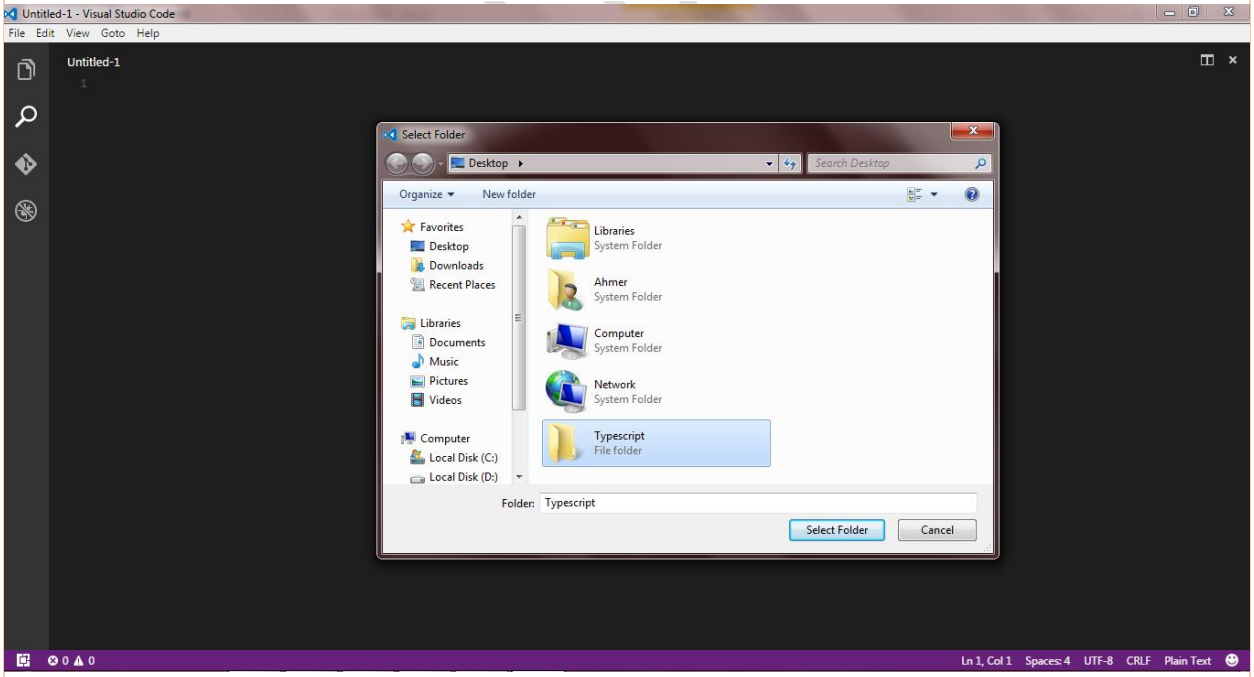
“LET’S GET STARTED”

Installing TS:

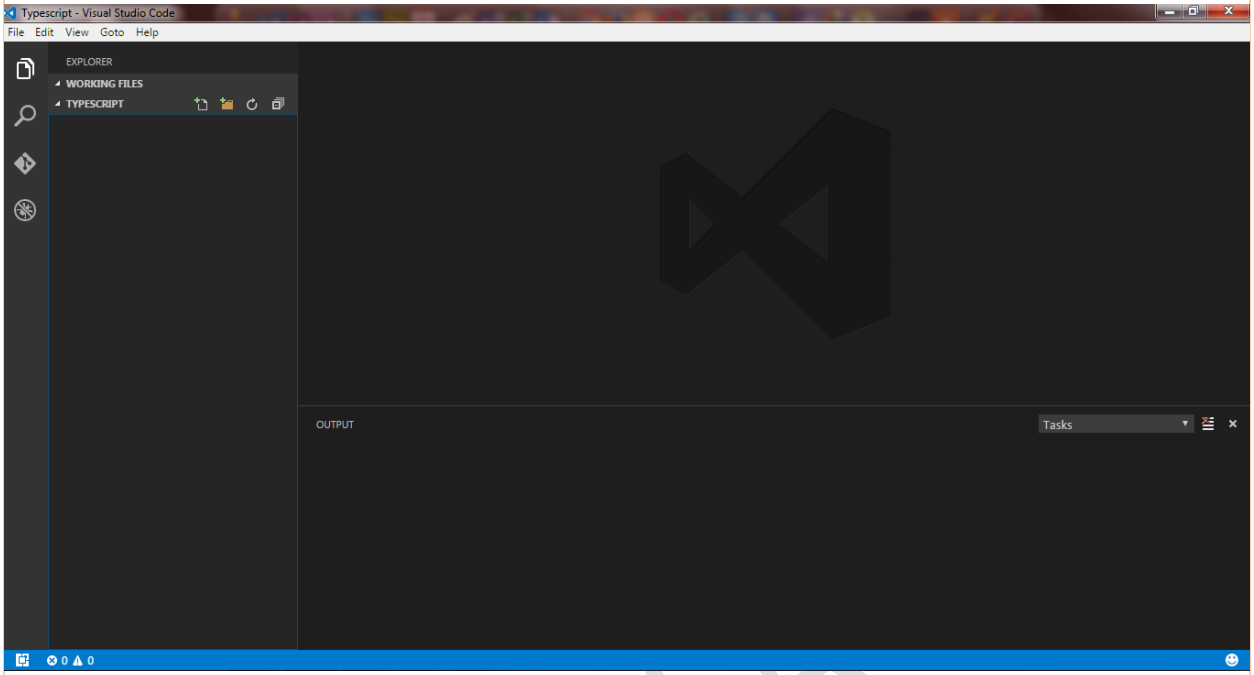
- Install Node.JS from below link
  - https://nodejs.org/en/
- Open command prompt and type below command. By using below command in CMD we ensure that is Node.js has completely installed in our PC.
  - **Node –v**
- Now install TS using below command in CMD
  - **npm install –g typescript**
- For checking is TS successfully installed in our PC we use below command in CMD
  - **Tsc –v**
- We are finally installed TS in our PC.
- Now, we need a code editor for learning TypeScript with best developer tooling experience. Download Visual Studio Code from below link, install it. It’s cross platform code editor which can be used in Linux, Mac OS and Windows OS.
  - https://code.visualstudio.com/Download

Trying Our First TS Code in VS Code:

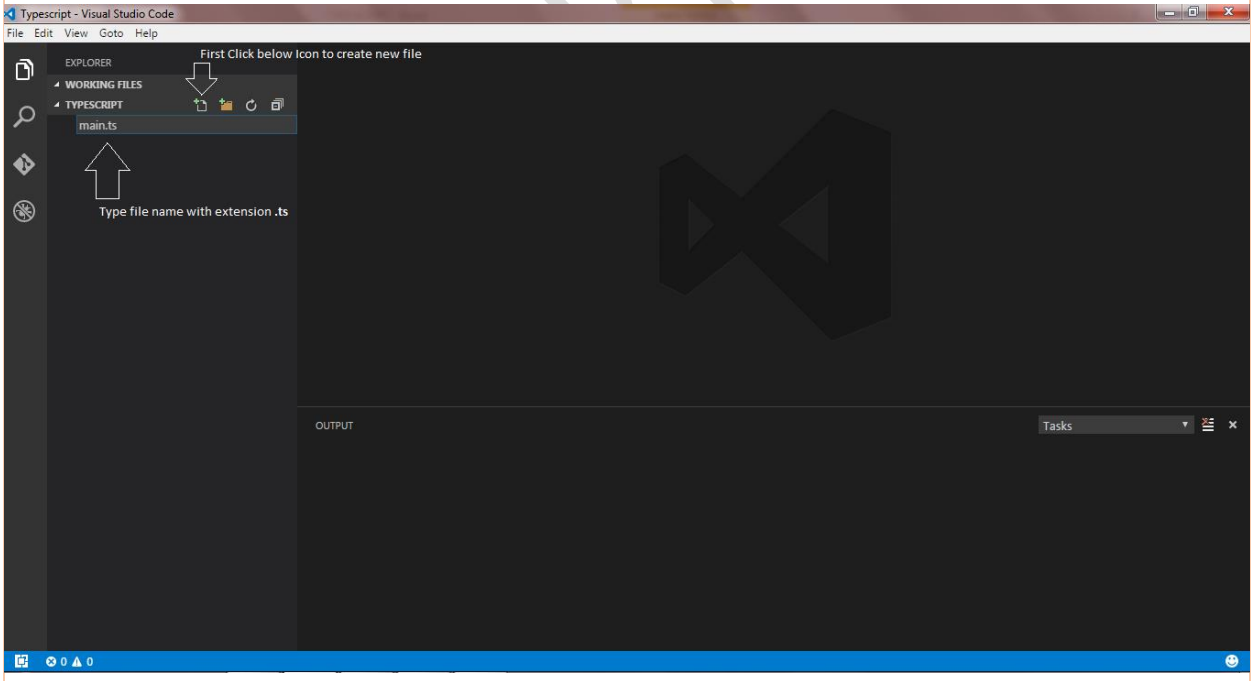
STEPS
1. Open VS Code
2. Click on **File** and select **Open Folder.** Select a folder where you want to create your project.



3. Your first screen may be like below



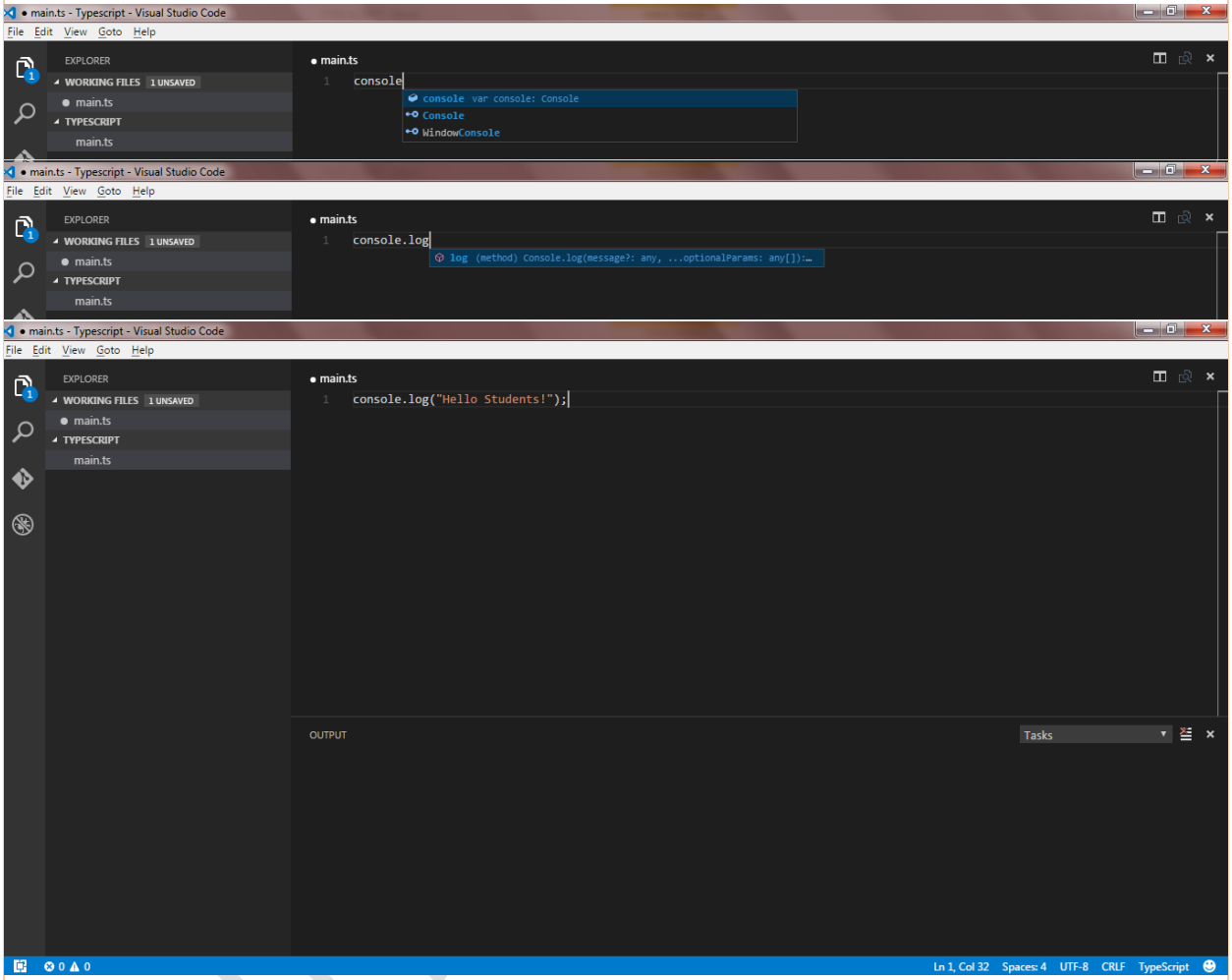
4. Click on highlighted button to create a new file named [**main.ts**] or Type your [**custom_name**]



5. Now come on right hand side. Type **console.log("Hello students!");** and Press CTRL + S to save your work.

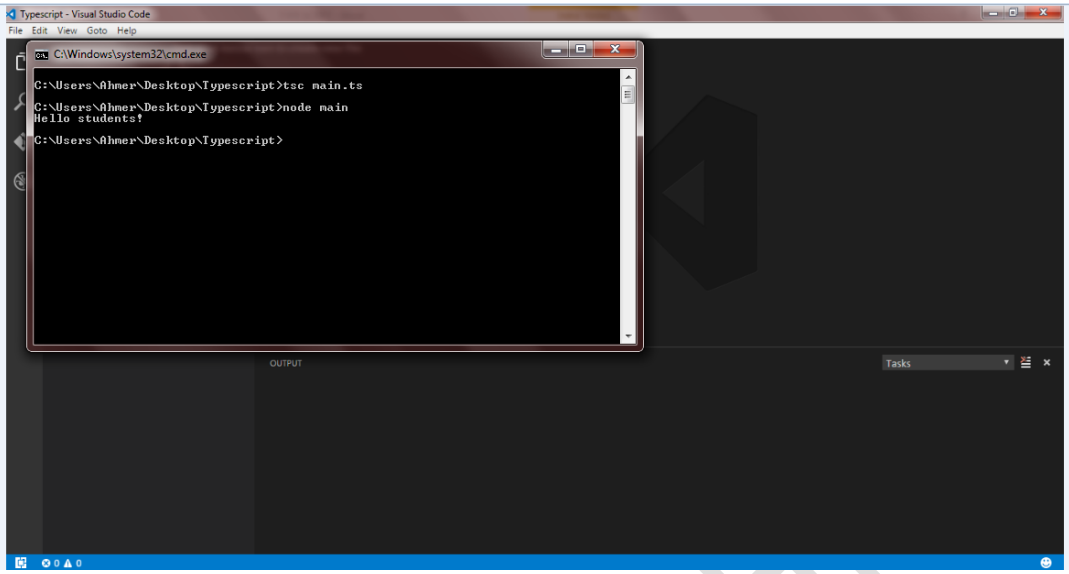
Note: VS Code provide us IntelliSense in below Languages

C++, CSS, HTML, JavaScript, JSON, Less, PHP, Python, Sass



6. In step six we are going to run above code in CMD. So we can ensure the output of our code is same as our expectations.
7. Open CMD and goto your project directory where you create [**main.ts]** or your custom [**.ts]** file and type **tsc main.ts** hit enter this will generate / convert your TS Code into JS Code.

8. Now type **node main** hit enter this will show you output of your JS Code.



Summary:

In above steps we learn How to:

- Installing Node.js
- Installing TypeScript
- Installing VS Code
- Get familiar with VS Code and create our first app [**main.ts**]
- How to check our output using CMD

Set Environment in VS Code:

VS Code is folder and file based. You can open a folder and work on its files. No project file. No solution file. Just grab the code folder and go. When there is a project context, such as with ASP.NET 5, and you open a folder (with an ASP.NET 5 project), VS Code detects the project context.

We had doing this till now. That we need to run the TS Complier every time to transpiling TS Code into JS manually in CMD. The **tsc main.ts** command will compile the TypeScript file into JavaScript and output into the same folder.

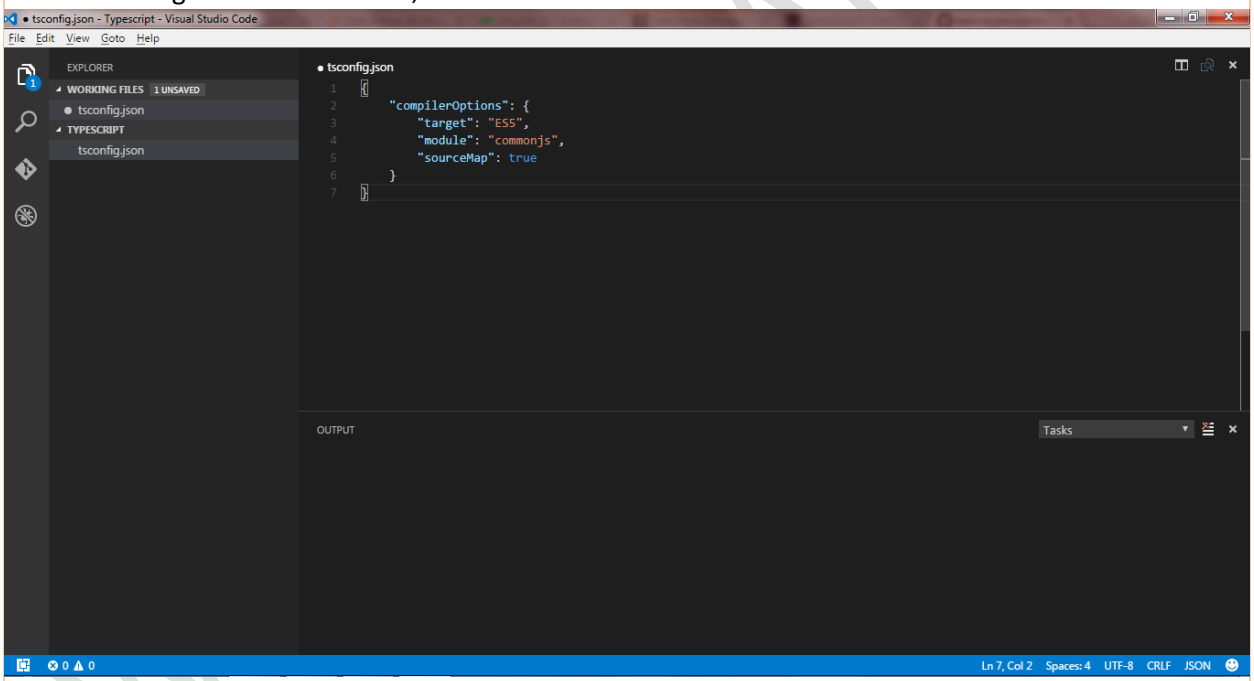
For some rich development features, automating the compilation or to prevent running above command every time. So we setting up the development environment in VS Code.

Before proceeding further more. We should discuss. That following are the features of rich development in VS Code:

- **Fast** - How fast this tool is. It opens fast. We can edit fast. We can debug fast. We can navigate fast.
- **Debugging** - Awesome, fast, and easy debugging of server side JavaScript and C#
- **IntelliSense** - You wouldn't want to be writing .NET code without the comforts of Visual Studio IntelliSense, right? VS Code knows this and tries to please with out-of-the-box IntelliSense. Sure you get friendly prompts on language features, but also smart IntelliSense that is local context aware in your custom code
- **Git integration** - super helpful to be able to integrate with git, show diffs, stage, commit, clean
- **Side-by-Side Editing** - VS Code can support up to three simultaneous file edits

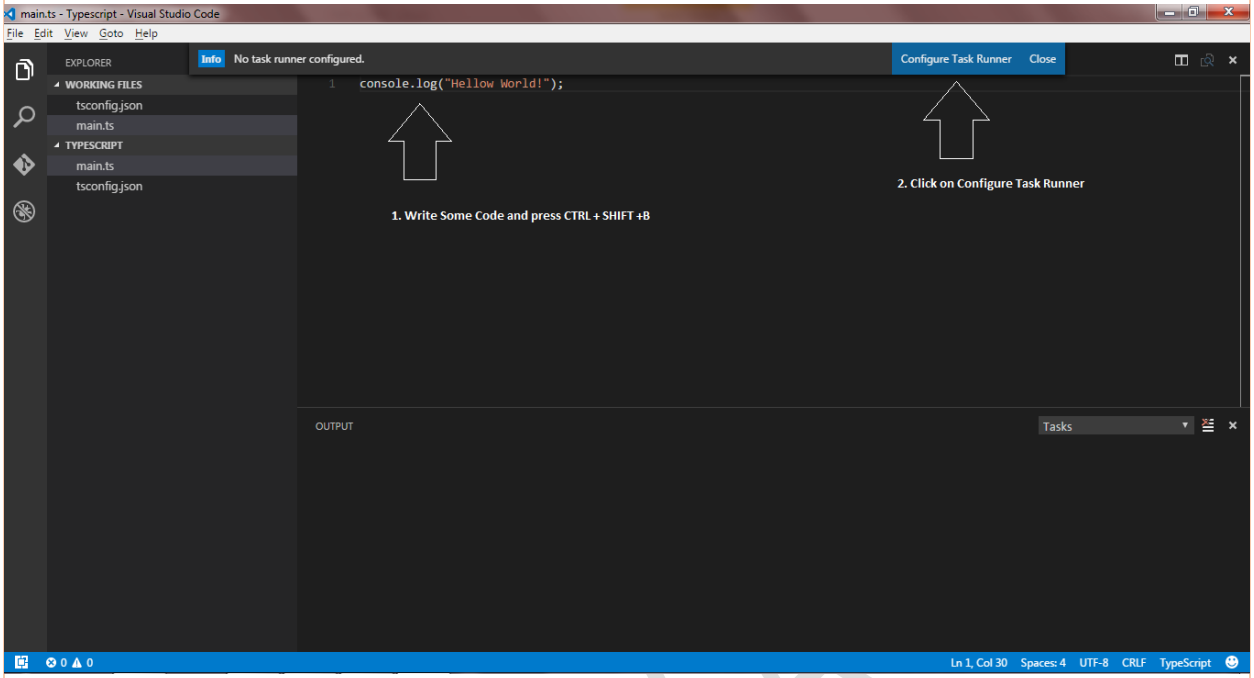
Let's have a look on how to setting up development environment in VS Code.

STEPS
1. Select your project folder
2. Create a [**tsconfig.json]** file on root
3. On right hand side editor, write below code

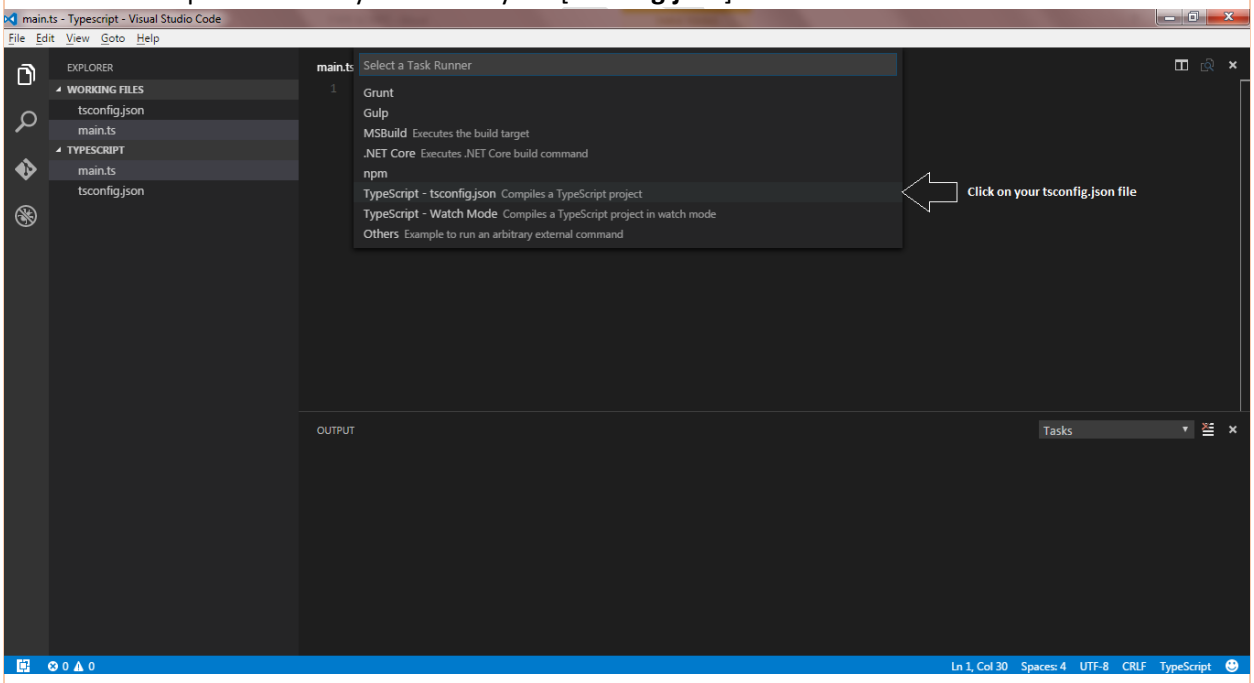


4. Create a [**main.ts**] file on root or create a new folder inside your project folder and in that you'll create your custom TS file.

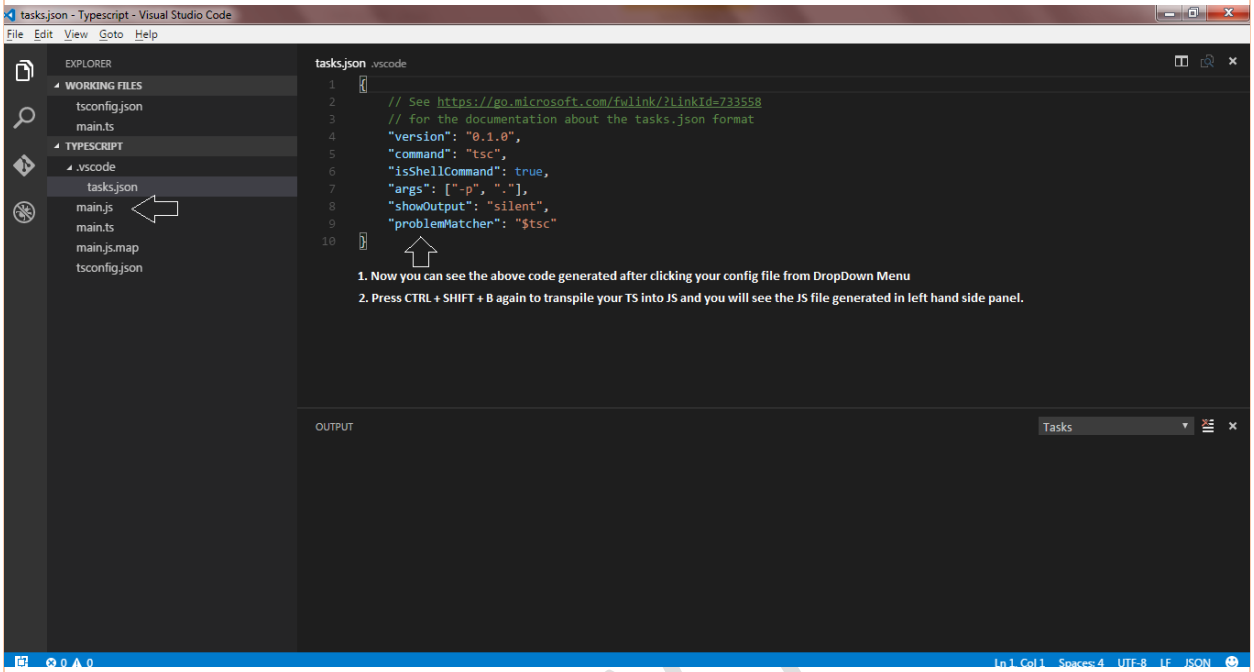
5. Write some code in [**main.ts**] and press **CTRL + SHIFT + B**



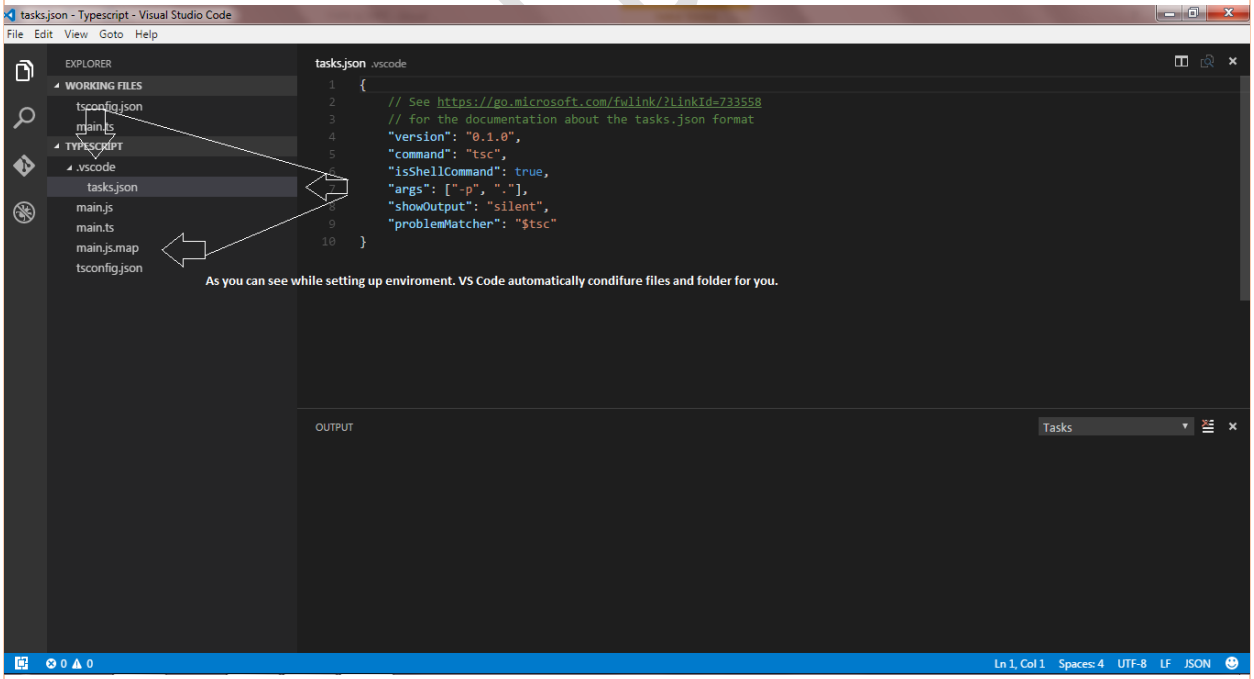
6. In Dropdown Menu you will see your [**tsconfig.json**] filename click on it



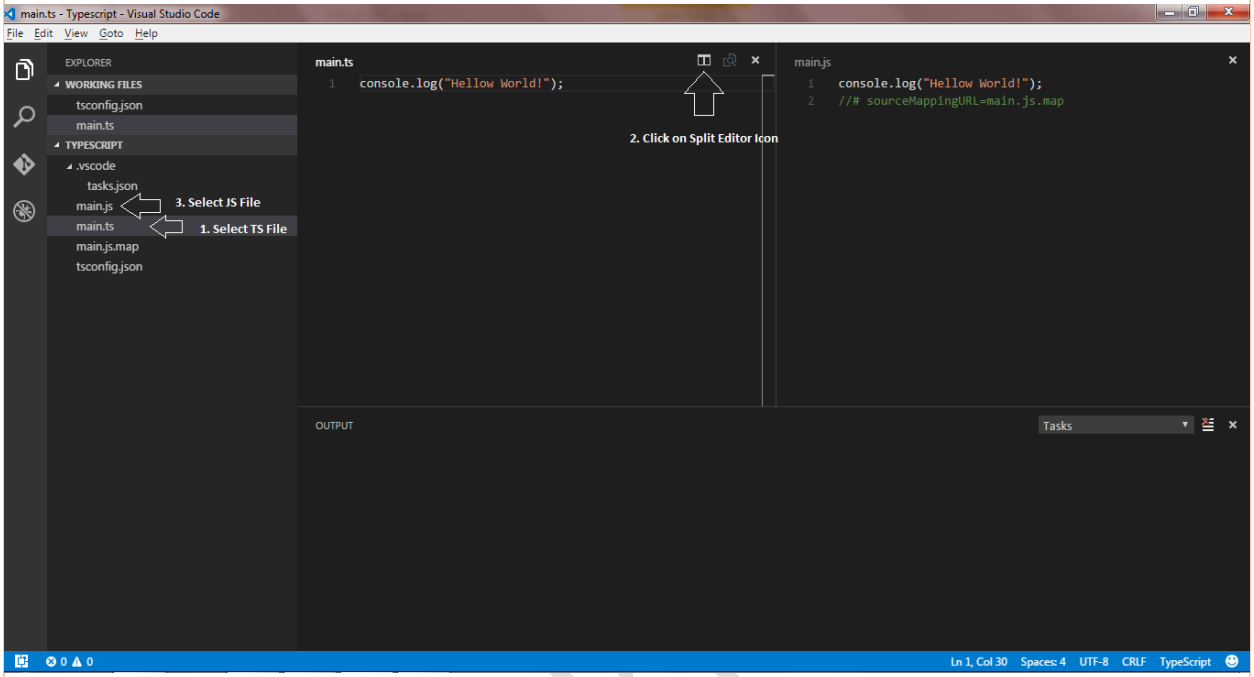
7. After clicking you'll see the auto generated code in your config file. Now you again press **CTRL + SHIFT + B** to transpile your TS into JS



8. As you'll see after setting up environment VS Code generated some files and folder automatically for you. This is indicate that you have successfully configure environment in VS Code



9. Checking environment in VS Code you select [**main.ts**] and click on Split Editor Icon on Top right of your screen and select [**main.js**] file. This will show you TS and JS code in separated editor. Now whenever you change or update code in your TS File and press **CTRL + SHIFT + B**. VS Code automatically transpile your code into JS Code and Show you in [**main.js**] editor



10. For check your output you either create an empty HTML Page or include your [**main.js**] in it. Or use CMD. Like we did it in our earlier steps.

We will learn TypeScript by taking small incremental steps. We are learning this language because Angular 2 is being built using TypeScript. We will use a combination of TypeScript and Angular 2 to build web apps. You can also start learning Angular 2 from: https://github.com/panacloud/learn-angular2

TABLE OF CONTENTS

00	HELLO WORLD

Code:

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

Description:
No Description for basic operations.

01	STRONG TYPING

Code:

TS	JS

//strongly typed syntax
var a : string = "Pakistan";
a = "USA";
var b : number = 9;
var c : boolean = true;
	//strongly typed syntax
	var a = "Pakistan";
	a = "USA";
	var b = 9;
	var c = true;

Description:
A strongly-typed programming language is one in which each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.

Advantages:

- The advantage of strongly typed languages is that the compiler can detect when an object is being sent a message to which it does not respond. This can prevent run-time errors.
- Earlier detection of errors speeds development
- Better optimized code from compiler
- No run-time penalty for determining type

Disadvantages:

- Loss of some flexibility
- More difficult to define collections of heterogeneous objects

Heterogeneous:
When a container class contains a group of mixed objects, the container is called a heterogeneous container.

02	CONST & LET

Code:

Const Example:

TS	JS

//use const where veriable values do not change
const a = 5;
const b : number = 33;
const c ="best";
	//use const where veriable values do not change
	var a = 5;
	var b = 33;
	var c = "best";

Let Example:

TS	JS

//Global Scope
//They are identical when used like this outside a function block.
let Name = "Ahmer Ali Ahsan"; //globally scoped
var Email = "ahmer.ali.ahsan@outlook.com"; //globally scoped
 	
//Function Scope
//They are identical when used like this in a function block.
function Example1() {
  let comment = "awesome worker!"; //function block scoped
  var available = "Fb!"; //function block scoped
}
 	
//Block Scope
//Here is the difference. let is only visible in the for() loop and var is visible to the whole function.
function Example2() {
  //[i] is *not* visible out here
 	
  for(let i = 0; i < 5; i++) {
  //[i] is only visible in here (and in the for() parentheses)
  }
 	
  //[i] is *not* visible out here
}
	//Global Scope
	//They are identical when used like this outside a function block.
	var Name = "Ahmer Ali Ahsan"; //globally scoped
	var Email = "ahmer.ali.ahsan@outlook.com"; //globally scoped
	//Function Scope
	//They are identical when used like this in a function block.
	function Example1() {
	  var comment = "awesome worker!"; //function block scoped
	  var available = "Fb!"; //function block scoped
	}
	//Block Scope
	//Here is the difference. let is only visible in the for() loop and var is visible to the whole function.
	function Example2() {
	  //[i] is *not* visible out here
	  for (var i = 0; i < 5; i++) {
	  }
	  //[i] is *not* visible out here
	}
	function Example3() {
	  //[j] *is* visible out here
	  for (var j = 0; j < 5; j++) {
	  }
	  //[j] *is* visible out here
	}

function Example3() {
  //[j] *is* visible out here
 	
  for(var j = 0; j < 5; j++) {
  //[j] is visible to the whole function
  }
 	
  //[j] *is* visible out here
}	

Description:

Const:

If you make any variable as constant, using **const** keyword, you cannot change its value. Also, the constant variables must be initialized while declared.

Let:

The difference is scoping. **var** is scoped to the nearest function block and **let** is scoped to the nearest enclosing block (both are global if outside any block), which can be smaller than a function block.

03 DUCK TYPING

Code:

TS	JS

//Create Object, Define and initialize its properties
let mytype = {
  id:1,
  name:"Ahmer"
};
 	
//Case1
//Re-use same properties
//Case 1: can only assign a value which has the the same properties
mytype = {
  id : 2,
  name: "Ali Ahsan"
};
 	
//Case 1a
//Error, renamed or missing property
	//Create Object, Define and initialize its properties
	var mytype = {
	  id: 1,
	  name: "Ahmer"
	};
	//Case1
	//Re-use same properties
	//Case 1: can only assign a value which has the the same properties
	mytype = {
	  id: 2,
	  name: "Ali Ahsan"
	};
	//Case 1a
	//Error, renamed or missing property
	//Object literals can only have properties that exist in contextual
	mytype = {

```
//Object literals can only have properties that exist in contextual
mytype = {
   id : 2,
   name_person: "Ali Ahsan"
};

//Case 1b
//Error, excess property
mytype = { id: 2,  name: "Ahmer", age: 24 };

//Case 1c
//Can only assign a type which has the the same properties and same rule
let myType2 = { id: 3,  name: "Ahmer" };
mytype = myType2;

//Case 1d
//Error, renamed or missing property and same rule
let myType3 = { id: 2,  name_person: "Azeem" };
mytype = myType3;



//Case 1e
//Ok, excess property allowed in case of stale object which is different from fresh object
var myType4 = { id: 2,  name: "Ahmer", age: 22 };
mytype = myType4;

//Case 2
//A type can include an index signature to explicitly indicate that excess properties are permitted in with fresh objects
var x: { id: number, [x: string]: any };
//Note now 'x' can have any name, just that the property should be of type string
x = { id: 1, FirstName: "Ahmer Ali" };  // Ok, 'fullname' matched by index signature

//Case 2a
//A type can include an index signature to explicitly indicate that excess properties are permitted in with fresh objects
```

```
   id: 2,
   name_person: "Ali Ahsan"
};
//Case 1b
//Error, excess property
mytype = { id: 2, name: "Ahmer", age: 24 };
//Case 1c
//Can only assign a type which has the the same properties and same rule
var myType2 = { id: 3, name: "Ahmer" };
mytype = myType2;
//Case 1d
//Error, renamed or missing property and same rule
var myType3 = { id: 2, name_person: "Azeem" };
mytype = myType3;
//Case 1e
//Ok, excess property allowed in case of stale object which is different from fresh object
var myType4 = { id: 2, name: "Ahmer", age: 22 };
mytype = myType4;
//Case 2
//A type can include an index signature to explicitly indicate that excess properties are permitted in with fresh objects
var x;
//Note now 'x' can have any name, just that the property should be of type string
x = { id: 1, FirstName: "Ahmer Ali" }; // Ok, 'fullname' matched by index signature
//Case 2a
//A type can include an index signature to explicitly indicate that excess properties are permitted in with fresh objects
x = { id: 2, FirstName: "Ahmer Ali", LastName: "Ahsan" };
//Case 3a
var y = { id: 1, fullname: "Ahmer Ali Ahsan" };
x = y; // Ok, 'FirstName' matched by index signature
y = x; // Error, Because "y" is a variable and "x" is a object with index signature
//Case 3b
var obj1;
```

```
x = { id: 2, FirstName: "Ahmer Ali", LastName: "Ahsan"};

//Case 3a
var y = {id: 1, fullname: "Ahmer Ali Ahsan"};

x = y; // Ok, 'FirstName' matched by index signature

y = x; // Error, Because "y" is a variable and "x" is a object with index signature

//Case 3b
var obj1 : {id:number, firstname:string};

var var1 = {id: 1, firstname: "Ahmer"};

obj1 = var1; //Ok, Because "obj1" and "var1" have same datatypes

//Case 3c
var obj2 : {id:number, firstname:string, middlename: string};

var var2 = {id: 1, firstname: "Ahmer"};

obj2 = var2; //Error, Because "obj2" and "var2" have different datatypes
```

```
var var1 = { id: 1, firstname: "Ahmer" };
obj1 = var1; //Ok, Because "obj1" and "var1" have same datatypes
//Case 3c
var obj2;
var var2 = { id: 1, firstname: "Ahmer" };
obj2 = var2; //Error, Because "obj2" and "var2" have different datatypes
//# sourceMappingURL=main.js.map
```


Description:

- Duck-Typing is a method/rule for checking the type compatibility for more complex variable types.
- TypeScript compiler uses the duck-typing method to compare one object with other object by comparing that the both objects have the same type matching properties/variables names or not. If both objects are different from one another and have different property names then the TypeScript compiler will generates the compile-time error through the duck-typing method/rule.
- Duck-typing feature gives type safety in TypeScript code.
- Through the duck-typing rule TypeScript compiler checks that an object is same as other object or not.
- According to duck-typing method, the both objects must have matching same properties/variables types.
- Duck-typing is a powerful feature which brings strong typing concepts in TypeScript code.

Object Literal:

JavaScript **Object Literal**. A JavaScript object literal is a comma-separated list of name-value pairs wrapped in curly braces. Object literals encapsulate data, enclosing it in a tidy package. This minimizes the use of global variables which can cause problems when combining code.

Index Signature:

A type can include an index signature to explicitly indicate that excess properties are permitted.

Example:

TS	JS

//index signature
//Object declaration & initialization
let obj1 : {id:number, name:string};
obj1 = {id: 1,name: "Ahmer Ali Ahsan"};
//Error, missing property name
obj1 = {id:1};
 	
let obj2 : {id:number, [obj2: string]:any};
obj2 = {id: 1, firstname: "Ahmer"};
//We add new property members due to index signatures
obj2 = {id: 2, firstname: "Ahmer", middlename:"Ali", lastname: "Ahsan"};
//We erase property members due to index signatures
obj2 = {id:2};
 	
//In below object, This is numeric index signature
//A numeric index signature says that properties with
//numeric name match a particular type.
//They have no effect on properties with non-numeric names.
let a :{[a: number]:boolean};
 	
a = {Name: "Ahmer"}; // OK! "Name" is not numeric
a = {Rollno: 69} // OK! "Rollno" is not numeric
	//index signature
	//Object declaration & initialization
	var obj1;
	obj1 = { id: 1, name: "Ahmer Ali Ahsan" };
	//Error, missing property name
	obj1 = { id: 1 };
	var obj2;
	obj2 = { id: 1, firstname: "Ahmer" };
	//We add new property members due to index signatures
	obj2 = { id: 2, firstname: "Ahmer", middlename: "Ali", lastname: "Ahsan" };
	//We erase property members due to index signatures
	obj2 = { id: 2 };
	//In below object, This is numeric index signature
	//A numeric index signature says that properties with
	//numeric name match a particular type.
	//They have no effect on properties with non-numeric names.
	var a;
	a = { Name: "Ahmer" }; // OK! "Name" is not numeric
	a = { Rollno: 69 }; // OK! "Rollno" is not numeric
	a = { lookThisIsNotNumeric: false }; // OK! "lookThisIsNotNumeric" is not numeric
	//Conversely, these are not OK, because the numeric-named properties do not fit the index signature type:

a = {lookThisIsNotNumeric: false} // OK! "lookThisIsNotNumeric" is not numeric
 	
//Conversely, these are not OK, because the numeric-named properties do not fit the index signature type:
 	
a = { 67: "hello" }; // Error! 67 is a number and "hello" is not a boolean
a = { 68: 653 }; // Error! 68 is a number and 653 is not a boolean
a = { 69: "hello" }; // Error! 69 is a number and "hello" is not a boolean
 	
//Finally, below code is OK, because the numeric-named property has a boolean value:
 	
a = { 70: true };
	a = { 67: "hello" }; // Error! 67 is a number and "hello" is not a boolean
	a = { 68: 653 }; // Error! 68 is a number and 653 is not a boolean
	a = { 69: "hello" }; // Error! 69 is a number and "hello" is not a boolean
	//Finally, below code is OK, because the numeric-named property has a boolean value:
	a = { 70: true };
	//# sourceMappingURL=main.js.map

04 ARRAYS

Code:

TS	JS

let array1: number[] = [1,2,3]; // Correct syntax
console.log(array1[1]);
 	
let array2: Array<number> = [3,4,5]; // Alternative correct syntax
console.log(array2[2]);
 	
let array3: number[] = []; // Correct syntax to define an empty array
array3.push(1234); // Insert new data into empty array
console.log(array3[0]);
 	
let array4: number[] = new number[3]; //Error
 	
let array5: number[] = ["Ahmer","Ali","Ahsan"]; // Error
	var array1 = [1, 2, 3]; // Correct syntax
	console.log(array1[1]);
	var array2 = [3, 4, 5]; // Alternative correct syntax
	console.log(array2[2]);
	var array3 = []; // Correct syntax to define an empty array
	array3.push(1234); // Insert new data into empty array
	console.log(array3[0]);
	var array4 = new number[3]; //Error
	var array5 = ["Ahmer", "Ali", "Ahsan"]; // Error

Description:

An array is simply marked with the [] notation, similar to JavaScript, and each array can be strongly typed to hold a specific type.

05	ANY TYPE

Code:

TS	JS

let myType : any = { name: "Ahmer", id: 1 };
 	
myType = { id: 2, name: "Stephan" };// can only assign a type which has the at least the same properties
 	
myType = { id: 3, name: "Alina", gender: false };//becuase of any it assign a different type
 	
myType = { name: "Ian Somerhalder", gender: false };//can even reduce the properties because of any type
 	
myType = "Even a sring can be assigned";
 	
myType = function(){ console.log("Even a function can be assigned to any")};
 	
let notSure: any = 4;
 	
notSure = "maybe a string instead";
 	
notSure = false;
	var myType = { name: "Ahmer", id: 1 };
	myType = { id: 2, name: "Stephan" }; // can only assign a type which has the at least the same properties
	myType = { id: 3, name: "Alina", gender: false }; //becuase of any it assign a different type
	myType = { name: "Ian Somerhalder", gender: false }; //can even reduce the properties because of any type
	myType = "Even a sring can be assigned";
	myType = function () { console.log("Even a function can be assigned to any"); };
	var notSure = 4;
	notSure = "maybe a string instead";
	notSure = false;

Description:

JavaScript is flexible enough to allow variables to be mixed and matched. The following code snippet is actually valid JavaScript code:

For Example:

```
var item1 = { id: 1, name: "item 1" };
item1 = { id: 2 };
```

Our first line of code assigns an object with an id property and a name property to the variable item1. The second line then re-assigns this variable to an object that has an id property but not a name property. Unfortunately, as we have seen previously, TypeScript will generate a compile time error for the preceding code:

error TS2012: Build: Cannot convert '{ id: number; }' to '{ id: number; name: string; }'

TypeScript introduces the any type for such occasions. Specifying that an object has a type of any in essence relaxes the compiler's strict type checking. The following code shows how to use the any type:

```
var item1 : any = { id: 1, name: "item 1" };
item1 = { id: 2 };
```

Note how our first line of code has changed. We specify the type of the variable item1 to be of type : any so that our code will compile without errors. Without the type specifier of : any, the second line of code, would normally generate an error.

06 EXPLICIT CASTING

Code:

TS	JS

//Example 1:
// Below is the object which have two properties types number
let a = {id: 1, Rollno: 100};
 	
//lets add a new property in "a" which type is string
a = {id: 1, Rollno: 101, Name: "Ahmer"};
//Error, object literal may only specify known properties and "Name" not exists in type
 	
//What can we do now? Here we use explicit casting using "any" type in "< >" syntax
a = <any> {id: 1, Rollno: 101, Name: "Ahmer"};
// Now by using explicit casting you change the type of an object and you erase or increase properties of an object
	//Example 1:
	// Below is the object which have two properties types number
	var a = { id: 1, Rollno: 100 };
	//lets add a new property in "a" which type is string
	a = { id: 1, Rollno: 101, Name: "Ahmer" };
	//Error, object literal may only specify known properties and "Name" not exists in type
	//What can we do now? Here we use explicit casting using "any" type in "< >" syntax
	a = { id: 1, Rollno: 101, Name: "Ahmer" };
	// Now by using explicit casting you change the type of an object and you erase or increase properties of an object
	//Example 2:
	//Below is the variable which type is number

//Example 2:
 	
//Below is the variable which type is number
var b = 10;
 	
//We can add string here by using <any>
b = <any> true;
 	
b = "karachi";
//Error, Because in above code we aren't use explicit casting.
	var b = 10;
	//We can add string here by using <any>
	b = true;
	b = "karachi";
	//Error, Because in above code we aren't use explicit casting.

Description:
Explicit provides conversion functionality. An explicit conversion involves casting from one type to another. We implement the casting functionality in TypeScript using < > syntax.

07	ENUM

Code:

TS	JS

//Create Enum Object
enum color {
  Red, //output 0
  Blue, //output 1
  Green //output 2
}
console.log(color.Red); //output "0"
console.log(color["Red"]); //output "0"
console.log(color[0]);
// In above code. We are getting the string "Red" – which is a string representation of our original enum name
// This allowing us to access a string representation of our enum value
 	
// Also you can change the number associated with any enum member by assigning to it specifically
enum color {
  yellow = 3, //output 3
  purple, //output 4
  skyblue //output 5
	//Create Enum Object
	var color;
	(function (color) {
	  color[color["Red"] = 0] = "Red";
	  color[color["Blue"] = 1] = "Blue";
	  color[color["Green"] = 2] = "Green"; //output 2
	})(color || (color = {}));
	console.log(color.Red); //output "0"
	console.log(color["Red"]); //output "0"
	console.log(color[0]);
	// In above code. We are getting the string "Red" – which is a string representation of our original enum name
	// This allowing us to access a string representation of our enum value
	// Also you can change the number associated with any enum member by assigning to it specifically
	var color;
	(function (color) {
	  color[color["yellow"] = 3] = "yellow";
	  color[color["purple"] = 4] = "purple";

}
console.log(color.yellow); //output "3"
console.log(color["yellow"]); //output "3"
console.log(color[3]); //output "yellow"
	color[color["skyblue"] = 5] = "skyblue";
	//output 5
	})(color || (color = {}));
	console.log(color.yellow); //output "3"
	console.log(color["yellow"]); //output "3"
	console.log(color[3]); //output "yellow"

Description:

Enums are a special type that has been borrowed from other languages such as C#, and provide a solution to the problem of special numbers. An enum associates a human-readable name for a specific number.

Note:

Javascript doesn't have enum data type. However TypeScript Does.

08 CONST ENUM

Code:

TS	JS

const enum color {
Red,
Blue,
Green
}
 	
console.log(color.Red); //output "0"
console.log(color["Red"]); //output "0"
console.log(color[0]);
 	
//With const enums, we therefore cannot reference the internal string value of an enum, as we did in our previous code sample. Consider the following example:
// generates an error
 	
console.log(color[0]);
 	
// valid usage
 	
console.log(color["Red"]);
	console.log(0 /* Red */); //output "0"
	console.log(0 /* "Red" */); //output "0"
	console.log(color[0]);
	//With const enums, we therefore cannot reference the internal string value of an enum, as we did in our previous code sample. Consider the following example:
	// generates an error
	console.log(color[0]);
	// valid usage
	console.log(0 /* "Red" */);

Description:

With the release of TypeScript 1.4, we are also able to define const enums.

When using const enums, just keep in mind that the compiler will strip away all enum definitions and simply substitute the numeric value of the enum directly into our JavaScript code.

09	FUNCTIONS

Code:

TS

```
//Named Function
function addNumbers(a: number, b: number):
number {

 return a + b;

};

//Anonymous function
let which_has_no_name = function (x: number =
4, y: number = 5): number {
                                        return x+y;
                                };

//Call anonymous function.
let Call_Anonymous_Function =
which_has_no_name(5,5);

// Anonymous Function with explict type or
Function Type
var myAdd1: (a:number, b:number) => number  =
function (a:number, b:number) : number {
   return (a + b);
};

console.log(myAdd1(5,6));

let myAdd2: (a:number, b:number) => number =
   function(x: number, y: number): number {
      return x + y;
   };

console.log(myAdd2(5,6));

// Lamda functions
var myAdd3 = (a : number, b : number) => a + b;
console.log(myAdd3(5 , 5));
```

JS

```
//Named Function
function addNumbers(a, b) {
   return a + b;
}
;
//Anonymous function
var which_has_no_name = function (x, y) {
   if (x === void 0) { x = 4; }
   if (y === void 0) { y = 5; }
   return x + y;
};
//Call anonymous function.
var Call_Anonymous_Function =
which_has_no_name(5, 5);
// Anonymous Function with explict type or
Function Type
var myAdd1 = function (a, b) {
   return (a + b);
};
console.log(myAdd1(5, 6));
var myAdd2 = function (x, y) {
   return x + y;
};
console.log(myAdd2(5, 6));
// Lamda functions
var myAdd3 = function (a, b) { return a + b; };
console.log(myAdd3(5, 5));
```

Description:

Functions:

Functions are the fundamental building block of any applications in JavaScript. TypeScript also adds some new capabilities to the standard JavaScript functions to make them easier to work with.

Named Functions:

A named function is a function that was declared with named identifier to refer it.

Anonymous Functions:

An anonymous function is a function that was declared without any named identifier to refer to it. As such, an anonymous function is usually not accessible after its initial creation.

Typed Function / Anonymous Functions with explicit type:

A function's type has the same two parts: the type of the arguments and the return type. When writing out the whole function type, both parts are required. We write out the parameter types just like a parameter list, giving each parameter a name and a type. This name is just to help with readability.

As long as the parameter types line up, it's considered a valid type for the function, regardless of the names you give the parameters in the function type.

The second part is the return type. We make it clear which is the return type by using a fat arrow (=>) between the parameters and the return type. As mentioned before, this is a required part of the function type, so if the function doesn't return a value, you would use void instead of leaving it off.

Lamda Functions:

A lambda expression is an anonymous function.
It's a shorthand that allows you to write a method in the same place you are going to use it. Especially useful in places where a method is being used only once, and the method definition is short. It saves you the effort of declaring and writing a separate method to the containing class.

10 FUNCTION WITH OPTIONAL PARAMETER

Code:

TS

```
function buildName(firstName: string, lastName?: string) : string {//Named function with optional parameters
   if (lastName)
      return firstName + " " + lastName;
   else
      return firstName;
}

var result1 = buildName("Ahmer");  //works correctly because last parameter is optional
//var result2 = buildName("Ahmer", "Ali", "Ahsan");  //error, too many parameters
var result3 = buildName("Ahmer Ali", "Ahsan");  //correct



//anonymous function type with optional parameters
var buildName1 : (firstName: string, lastName?: string) => string =
function(firstName: string, lastName?: string) : string {
   if (lastName)
      return firstName + " " + lastName;
   else
      return firstName;
}
```

JS

```
function buildName(firstName, lastName) {
   if (lastName)
      return firstName + " " + lastName;
   else
      return firstName;
}
var result1 = buildName("Ahmer"); //works correctly because last parameter is optional
//var result2 = buildName("Ahmer", "Ali", "Ahsan");  //error, too many parameters
var result3 = buildName("Ahmer Ali", "Ahsan");  //correct
//anonymous function type with optional parameters
var buildName1 = function (firstName, lastName) {
   if (lastName)
      return firstName + " " + lastName;
   else
      return firstName;
};
```

Description:

Function parameters are required or that they are optional.

11 FUNCTION WITH DEFAULT PARAMETERS

Code:

TS

```
//Named function with optional and default
parameters
function buildName(firstName: string, lastName =
"Ahsan") : string {
   if (lastName)
      return firstName + " " + lastName;
   else
      return firstName;
}

var result1 = buildName("Ahmer Ali");
//works correctly because last parameter is
optional
//var result2 = buildName("Ahmer", "Ali",
"Ahsan");  //error, too many parameters
var result3 = buildName("Ahmer Ali ", "Ahsan");
//correct



//anonymous function type with defult
parameters (Note that the parameter type will be
optional when used with defult value)
var buildName1 : (firstName: string, lastName?:
string) => string =
function(firstName: string, lastName = "Ahsan") :
string {
   if (lastName)
      return firstName + " " + lastName;
   else
      return firstName;
}
```

JS

```
//Named function with optional and default
parameters
function buildName(firstName, lastName) {
   if (lastName === void 0) { lastName = "Ahsan"; }
   if (lastName)
      return firstName + " " + lastName;
   else
      return firstName;
}
var result1 = buildName("Ahmer Ali");
//works correctly because last parameter is
optional
//var result2 = buildName("Ahmer", "Ali",
"Ahsan");  //error, too many parameters
var result3 = buildName("Ahmer Ali ", "Ahsan");
//correct
//anonymous function type with defult
parameters (Note that the parameter type will be
optional when used with defult value)
var buildName1 = function (firstName, lastName)
{
   if (lastName === void 0) { lastName = "Ahsan"; }
   if (lastName)
      return firstName + " " + lastName;
   else
      return firstName;
};
```

Description:

Default function parameters allow formal parameters to be initialized with default values if no value or undefined is passed.

12 FUNCTION WITH REST PARAMETER

Code:

TS	JS

function buildName(firstName: string, ...restOfName: string[]) {//Named function with Rest parameters
return firstName + " " + restOfName.join(" ");
}
 	
var employeeName = buildName("Ahmer", "Ali", "Ahsan", "BlaBla");
 	
function testParams(...argArray: number[]) {
 	
if (argArray.length > 0) {
 	
for (var i = 0; i < argArray.length; i++) {
console.log("argArray " + i + " = " + argArray[i]);
console.log("arguments " + i + " = " + arguments[i]);
}
 	
}
 	
}
testParams(1);
 	
testParams(1, 2, 3, 4);
 	
testParams("one", "two");
//The last line in this sample will, however, generate a compile error, as we have
// defined the rest parameter to only accept numbers, and we are attempting to call
// the function with strings.
 	
//anonymous function type with Rest parameters
var buildNameFun: (fname: string, ...rest: string[])=>string =
function (firstName: string, ...restOfName: string[]) {
	function buildName(firstName) {
	var restOfName = [];
	for (var _i = 1; _i < arguments.length; _i++) {
	restOfName[_i - 1] = arguments[_i];
	}
	return firstName + " " + restOfName.join(" ");
	}
	var employeeName = buildName("Ahmer", "Ali", "Ahsan", "BlaBla");
	function testParams() {
	var argArray = [];
	for (var _i = 0; _i < arguments.length; _i++) {
	argArray[_i - 0] = arguments[_i];
	}
	if (argArray.length > 0) {
	for (var i = 0; i < argArray.length; i++) {
	console.log("argArray " + i + " = " + argArray[i]);
	console.log("arguments " + i + " = " + arguments[i]);
	}
	}
	}
	testParams(1);
	testParams(1, 2, 3, 4);
	testParams("one", "two");
	//The last line in this sample will, however, generate a compile error, as we have
	// defined the rest parameter to only accept numbers, and we are attempting to call
	// the function with strings.
	//anonymous function type with Rest parameters
	var buildNameFun = function (firstName) {
	var restOfName = [];
	for (var _i = 1; _i < arguments.length; _i++) {
	restOfName[_i - 1] = arguments[_i];
	}
	return firstName + " " + restOfName.join(" ");
	};

return firstName + " " + restOfName.join(" "); } 	

Description:

The rest parameter syntax allows us to represent an indefinite number of arguments as an array. To implement rest parameter in arguments of a function we use the three dots (…) syntax.

Difference between rest parameters and the arguments object:

There are three main differences between rest parameters and the arguments object:

- rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function
- the arguments object is not a real array, while rest parameters are Array instances, meaning methods like sort, map, forEach or pop can be applied on it directly
- the arguments object has additional functionality specific to itself (like the callee property).

13 LAMBDA

Code:

TS	JS

//Example1 : Simple lambda function takes 2 arguments and return sum of arguments
var Example1 = (a: number, b: number) => {
  return a + b;
}
 	
//Example2 : Simple lambda function takes 2 arguments and return sum of arguments
//Example2 : output var add1 = function(x, y){return x + y};
var Example2 = (x: number, y: number) => (x + y);
 	
//automatically creating the that-equals-this pattern
var Example3 = f => { return this.x = "x"; }
 	
var Example4 = f => { return this.x = f; }
 	
var Example5 = () => {
	var _this = this;
	//Example1 : Simple lambda function takes 2 arguments and return sum of arguments
	var Example1 = function (a, b) {
	  return a + b;
	};
	//Example2 : Simple lambda function takes 2 arguments and return sum of arguments
	//Example2 : output var add1 = function(x, y){return x + y};
	var Example2 = function (x, y) { return (x + y); };
	//automatically creating the that-equals-this pattern
	var Example3 = function (f) { return _this.x = "x"; };
	var Example4 = function (f) { return _this.x = f; };
	var Example5 = function () {
	  return _this.a = "Ali";
	};
	console.log(Example3("Ahmer")); //return "x"

```
        return this.a = "Ali";
}

console.log(Example3("Ahmer")); //return "x"
console.log(Example4("Ahmer")); //return
"Ahmer"
console.log(Example5()); //return "Ali"
```

```
console.log(Example4("Ahmer")); //return
"Ahmer"
console.log(Example5()); //return "Ali"
```

Desciption:

A lambda expression is an anonymous function.

It's a shorthand that allows you to write a method in the same place you are going to use it. Especially useful in places where a method is being used only once, and the method definition is short. It saves you the effort of declaring and writing a separate method to the containing class.

Note:

What is “this” Context?

Context is most often determined by how a function is invoked. When a function is called as a method of an object, this is set to the object the method is called on.

14.1 CALLBACKS TYPE

Code:

TS	JS

//Example #1
function myCallBack1(text) {
 	
console.log("inside myCallback " + text);
 	
}
 	
function callingFunction1(initialText, callback) {
console.log("inside CallingFunction");
callback(initialText);
}
callingFunction1("myText", myCallBack1);
 	
//Example #2
function myCallBack2(SomeNumber:number) {
 	
console.log("inside myCallback " + SomeNumber);
 	
}
function callingFunction2(initialParameter: number,callback: (text: number) => void)
{
callback(initialParameter);
}
 	
callingFunction2(2016, myCallBack2);
	//Example #1
	function myCallBack1(text) {
	console.log("inside myCallback " + text);
	}
	function callingFunction1(initialText, callback) {
	console.log("inside CallingFunction");
	callback(initialText);
	}
	callingFunction1("myText", myCallBack1);
	//Example #2
	function myCallBack2(SomeNumber) {
	console.log("inside myCallback " + SomeNumber);
	}
	function callingFunction2(initialParameter, callback) {
	callback(initialParameter);
	}
	callingFunction2(2016, myCallBack2);

Description:

A "callback" is any function that is called by another function which takes the first function as a parameter. A lot of the time, a "callback" is a function that is called when something happens. That something can be called an "event" in programmer-speak.

14.2 FUNCTION CALLBACKS AND SCOPE

Code:

TS	JS

function testScope() {
var testVariable = "myTestVariable";
function print() {
console.log(testVariable);
}
}
console.log(testVariable); //Error
	function testScope() {
	var testVariable = "myTestVariable";
	function print() {
	console.log(testVariable);
	}
	}
	console.log(testVariable); //Error

Description:

JavaScript uses lexical scoping rules to define the valid scope of a variable. This means that the value of a variable is defined by its location within the source code. Nested functions have access to variables that are defined in their parent scope.

As an example of this, consider the Above TypeScript code:

This code snippet defines a function named testScope. The variable testVariable is defined within this function. The print function is a child function of testScope, so it has access to the testVariable variable. The last line of the code, however, will generate a compile error, because it is attempting to use the variable testVariable, which is lexically scoped to be valid only inside the body of the testScope function:

error TS2095: Build: Could not find symbol 'testVariable'.

Simple, right? A nested function has access to variables depending on its location within the source code. This is all well and good, but in large JavaScript projects, there are many different files and many areas of the code are designed to be re-usable.

15 FUNCTIONS OVERLOAD

Code:

TS	JS

function add(arg1: string, arg2: string): string;//option 1
function add(arg1: number, arg2: number): number;//option 2
function add(arg1: boolean, arg2: boolean): boolean;//option 3
function add(arg1: any, arg2: any): any {//this is not part of the overload list, so it has only three overloads
  return arg1 + arg2;
}
 	
//Calling 'add' with any other parameter types would cause an error except for the three options
console.log(add(1, 2));
console.log(add("Hello", "World"));
console.log(add(true, false));
	function add(arg1, arg2) {
	  return arg1 + arg2;
	}
	//Calling 'add' with any other parameter types would cause an error except for the three options
	console.log(add(1, 2));
	console.log(add("Hello", "World"));
	console.log(add(true, false));

Description:

As JavaScript is a dynamic language, we can often call the same function with different argument types. Consider above example.

For Example:

```
function add(x, y) {
        return x + y;
}
console.log("add(1,1) = " + add(1,1));
console.log("add(''1'',''1'') = " + add("1", "1"));
console.log("add(true,false) = " + add(true, false));
```

Here, we are defining a simple add function that returns the sum of its two parameters, x and y.
If we run this code, we will see the following output:

add(1,1) = 2
add('1','1') = 11
add(true,false) = 1

TypeScript introduces a specific syntax to indicate multiple function signatures for the same function. If we were to replicate the preceding code in TypeScript, we would need to use the function overload syntax:

```
function add(arg1: string, arg2: string): string;//option 1
function add(arg1: number, arg2: number): number;//option 2
function add(arg1: boolean, arg2: boolean): boolean;//option 3
function add(arg1: any, arg2: any): any {//this is not part of the overload list, so it has only three overloads
  return arg1 + arg2;
}
```

16 UNION TYPES

Code:

TS	JS

function addWithUnion(arg1: string | number | boolean, arg2: string | number | boolean): string | number | boolean {
  if (typeof arg1 === "string") {///This is known as a type guard and means that the type of x will be treated as a string within the if statement block
  // arg1 is treated as a string here
  return arg1 + "is a string";
  }
  if (typeof arg1 === "number") {
  // arg1 is treated as a number here
  return arg1 + 10;
  }
  if (typeof arg1 === "boolean") {
  // arg1 is treated as a boolean here
  return arg1 && false;
  }
}
 	
function f(x: number | number[]) {
 if (typeof x === "number") { //This is known as a type guard and means that the type of x will be treated as a number within the if statement block
  return x + 10;
 }
 else {
  // return sum of numbers
 }
}
	function addWithUnion(arg1, arg2) {
	  if (typeof arg1 === "string") {
	  // arg1 is treated as a string here
	  return arg1 + "is a string";
	  }
	  if (typeof arg1 === "number") {
	  // arg1 is treated as a number here
	  return arg1 + 10;
	  }
	  if (typeof arg1 === "boolean") {
	  // arg1 is treated as a boolean here
	  return arg1 && false;
	  }
	}
	function f(x) {
	  if (typeof x === "number") {
	  return x + 10;
	  }
	  else {
	  }
	}
	/*Note on Type Guards:
	A common pattern in JavaScript is to use typeof or instanceof to examine the type of an expression at runtime.
	TypeScript now understands these conditions and will change type inference accordingly when used in an if block.
	This is called a type guard.*/

```
/*Note on Type Guards:
A common pattern in JavaScript is to use typeof
or instanceof to examine the type of an
expression at runtime.
TypeScript now understands these conditions and
will change type inference accordingly when used
in an if block.
This is called a type guard.*/

var x: any = "Tom"; //Line A
if(typeof x === 'string') { //Line B
  console.log(x.lengthX);// Error, 'lengthX' does
not exist on 'string' but 'lenght' does
}
// x is still any here
x.unknown();// OK
```

```
var x = "Tom"; //Line A
if (typeof x === 'string') {
   console.log(x.lengthX); // Error, 'lengthX' does
not exist on 'string' but 'lenght' does
}
// x is still any here
x.unknown(); // OK
```

Description:

With the release of TypeScript 1.4, we now have the ability to combine one or two types using the pipe symbol (|) to denote a Union Type.
Union types are a powerful way to express a value that can be one of several types.

Type Guards:

Within the body of the addWithUnion function in the preceding code snippet, we check whether the type of the arg1 argument is a string, with the statement typeof arg1 === "string". This is known as a type guard and means that the type of arg1 will be treated as a string within the if statement block. Within the body of the next if statement, the type of arg1 will be treated as a number, allowing us to add 10 to its value, and in the body of the last if statement, the type will be treated as a boolean by the compiler.

17 CUSTOM TYPE GUARD

Code:

TS	JS

//User defined type guards in 1.6
 	
//In earlier versions of TypeScript, you could use if statements to narrow the type. For example, you could use:
 	
//if (typeof x === "number") { … }
 	
//This helped type information flow into common ways of working with types at runtime (inspired.
//While this approach is powerful, TypeScript has now pushed it a bit further.
//In 1.6, you can now create your own type guard functions:
 	
interface Animal {name: string; }
interface Cat extends Animal { meow(); }
 	
function isCat(a: Animal): a is Cat {
return true;
}
 	
var x: Animal;
 	
if(isCat(x)) {
x.meow(); // OK, x is Cat in this block
}
 	
//This allows you to work with not only typeof and instanceof checks, which need a type that JavaScript understands,
//but now you can work with interfaces and do custom analysis. Guard functions are denoted by their “a is X” return type,
//which returns boolean and signals to the compiler if what the expected type now is.
	//User defined type guards in 1.6
	function isCat(a) {
	return true;
	}
	var x;
	if (isCat(x)) {
	x.meow(); // OK, x is Cat in this block
	}
	//This allows you to work with not only typeof and instanceof checks, which need a type that JavaScript understands,
	//but now you can work with interfaces and do custom analysis. Guard functions are denoted by their “a is X” return type,
	//which returns boolean and signals to the compiler if what the expected type now is.

Description:

In earlier versions of TypeScript, you could use if statements to narrow the type. For example, you could use:

Example:

```
if (typeof x === "number") { … }
```

This helped type information flow into common ways of working with types at runtime (inspired by some of the other projects doing type checking of JS). While this approach is powerful, we wanted to push it a bit further. In 1.6, you can now create your own type guard functions.

18 TYPE ALIASES

Code:

TS	JS

type StringNumberOrBoolean = string | number | boolean;
function addWithAliases(
arg1: StringNumberOrBoolean,
arg2: StringNumberOrBoolean
): StringNumberOrBoolean {
return arg1;
}
	function addWithAliases(arg1, arg2) {
	return arg1;
	}

Description:

We are also able to define an alias for a type, a union type, or a function definition. Type aliases are denoted by using the type keyword.

In above code, we have defined a type alias named StringNumberOrBoolean that is a type union of the string, number, and boolean types.

19 TUPLES

Code:

TS	JS

var tuple: [number, string] = [1, "bob"];
var firstElement = tuple[1]; // firstElement now has type 'number'
var secondElement = tuple[1,"Ahmer"]; // secondElement now has type 'string'
	var tuple = [1, "bob"];
	var firstElement = tuple[1]; // firstElement now has type 'number'
	var secondElement = tuple[1, "Ahmer"]; // secondElement now has type 'string'

Description:

Tuple types have the advantage that you can accurately describe the type of an array of mixed types.