CitiusTech

TypeScript

Version 1.0
March 2019

CitiusTech has prepared the content contained in this document based on information and knowledge that it reasonably believes to be reliable. Any recipient may rely on the contents of this document at its own risk and CitiusTech shall not be responsible for any error and/or omission in the preparation of this document. The use of any third party reference should not be regarded as an indication of an endorsement, an affiliation or the existence of any other kind of relationship between CitiusTech and such third party

Agenda

- **Introduction to TypeScript**
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- References

Introduction to TypeScript

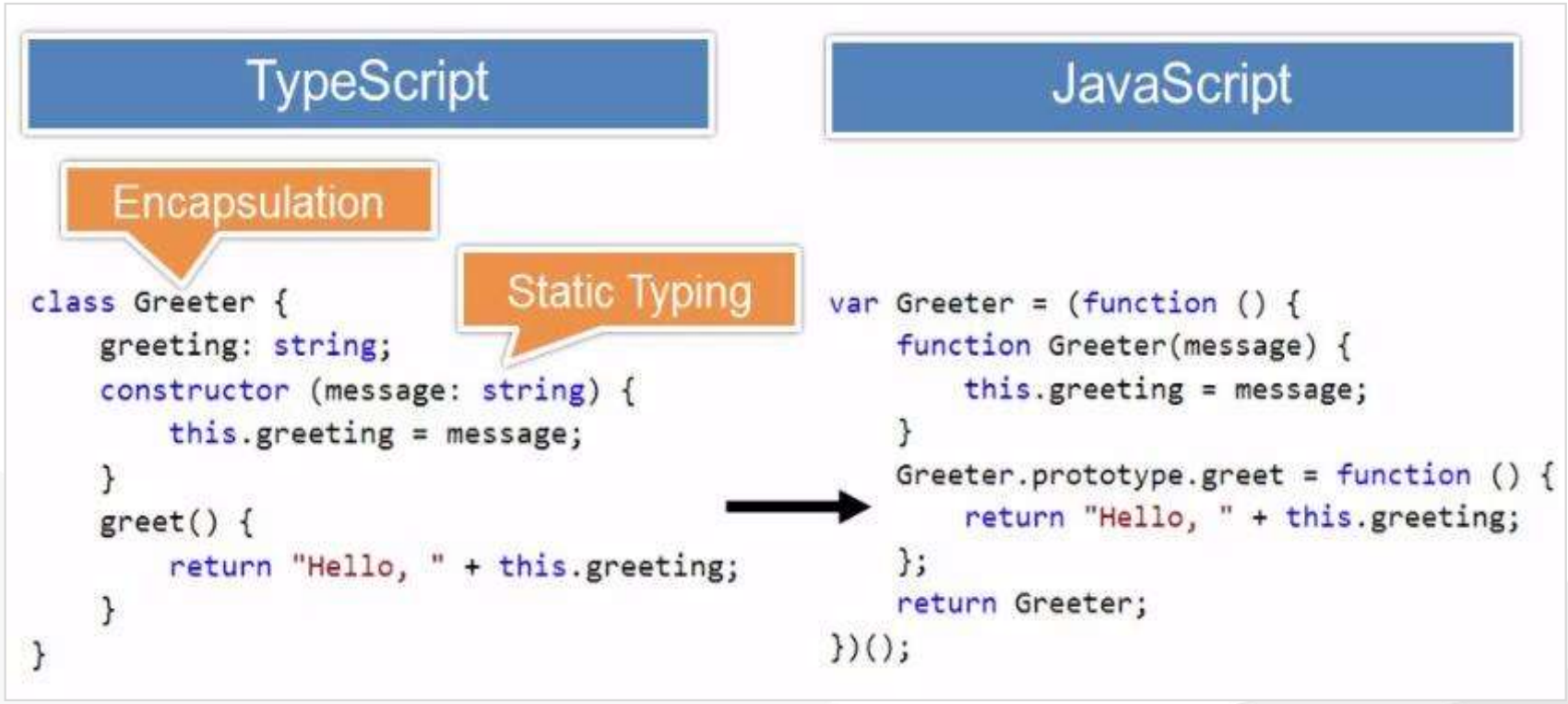
- TypeScript is a superset of JavaScript. Its emphasis on strong-typing & language constructs help us write client & server-side code with fewer errors
- TypeScript is cross-platform, open source & compiles to JavaScript that is compatible with every major browser & popular JavaScript frameworks.
- It's a strongly-typed language that compiles to Plain Old JavaScript which can be executed in any web browser or on a server in a Node JS application.
- TypeScript is the recommended language for developing Angular2 applications & also works great with other frameworks & libraries like React JS & jQuery.
- So, whether you are working with popular client-side frameworks, building server-side Node applications or just want a better version of JavaScript, TypeScript is the answer.

Agenda

- Introduction to TypeScript
- **TypeScript Compiler**
- Installing TypeScript
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- References

TypeScript Compiler

- Write TypeScript code and save it in a file with the .ts extension.
- Use TypeScript Compiler (tsc.exe), passing the .ts file as an argument to compile the Typescript code.
- This will be useful if the tool does not support TypeScript directly.
- The output of the compilation is a plain JavaScript file.



TypeScript Compiler Options (1/4)

Options which can be used with "tsc.exe":

- --module OR –m
  - Specifies the module format to be outputted by the compiler. This can be CommonJS or AMD

```
E:\>tsc --module CommonJS test.ts
    E:\>tsc --module AMD test.ts
```

- --target OR --t
  - Specifies the version of JavaScript to be outputted by the compiler. Default is ES3, but you can use ES5 as well.
- --watch OR --w
  - Leaves the compiler running in watch mode.
  - When changes are made to one of the source TypeScript files, the compiler automatically compiles it to JavaScript.

TypeScript Compiler Options (2/4)

- --outDir
  - Specifies a directory where the generated JavaScript files must be stored.
- --noImplicitAny
  - When used, & if the compiler infers that the type of some variable is any, then a compiler error occurs.
  - You can still use the any type, but it needs to be defined explicitly.
- The tsconfig.json file
  - Support was added in TypeScript 1.5
  - Marks the root of a TypeScript project.
  - Settings specified in this file will apply to the TypeScript files in that directory and all sub-directories.
  - When the compiler is started with no input files, the compiler looks for a file named tsconfig.json in the current directory & continues up the directory hierarchy until it finds parent directory.

TypeScript Compiler Options (3/4)

- This file can be used to specify compiler options.
  - When invoked without any compiler options, it uses the options specified in the tsconfig.json file.
  - Any options explicitly specified on the command-line will override the options specified in the tsconfig.json file.
- This also allows you to specify which files to include or exclude from your project.

```
{
   "compilerOptions" : {
      "target" : "es5",
      "outDir" : "js"
   },
   "files" : [
      "app.ts",
      "classes.ts"
   ]
}
```

TypeScript Compiler Options (4/4)

- Setting a top-level property compileOnSave signals to the IDE to generate all files for a given tsconfig.json upon saving.

```
{
   "compileOnSave" : true,
      "compilerOptions" : {
         "noImplicitAny" : true
      }
}
```

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- **Installing TypeScript**
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- References

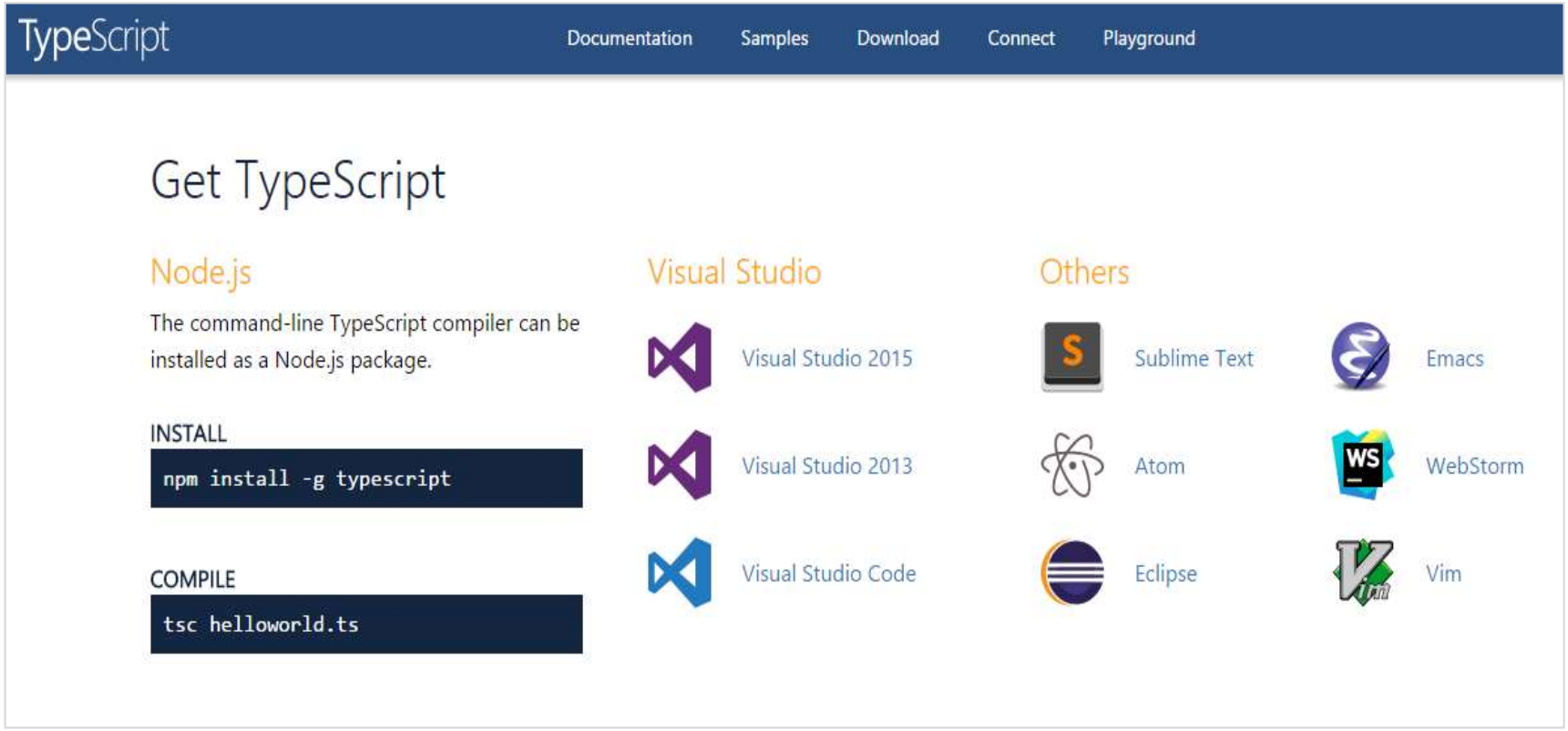
Installing TypeScript (1/2)

- There are two approaches to Installing TypeScript:
  - Install an extension to Visual Studio in case you want to use Visual Studio to write TypeScript
  - Install using npm

```
npm install -g typescript
```

Installing TypeScript (2/2)

- Install an extension to Visual Studio:



- Click on Visual Studio 2013 for installing an extension to VS2013
- One of the benefits of writing TypeScript in Visual Studio is that with no additional configuration, it will automatically compile the TypeScript file each time the file is saved.

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- **TypeScript Language Basics**
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- References

TypeScript Language Basics (1/8)

- All of the standard control structures found in JavaScript are immediately available within a TypeScript program. This includes :
  - Control flows
  - Data Types
  - Operators
  - Subroutines

{ } define code blocks.

Semi-colons end code expressions.

TypeScript Language Basics (2/8)

Important keywords and operators:

TypeScript is JavaScript with high-level language features.

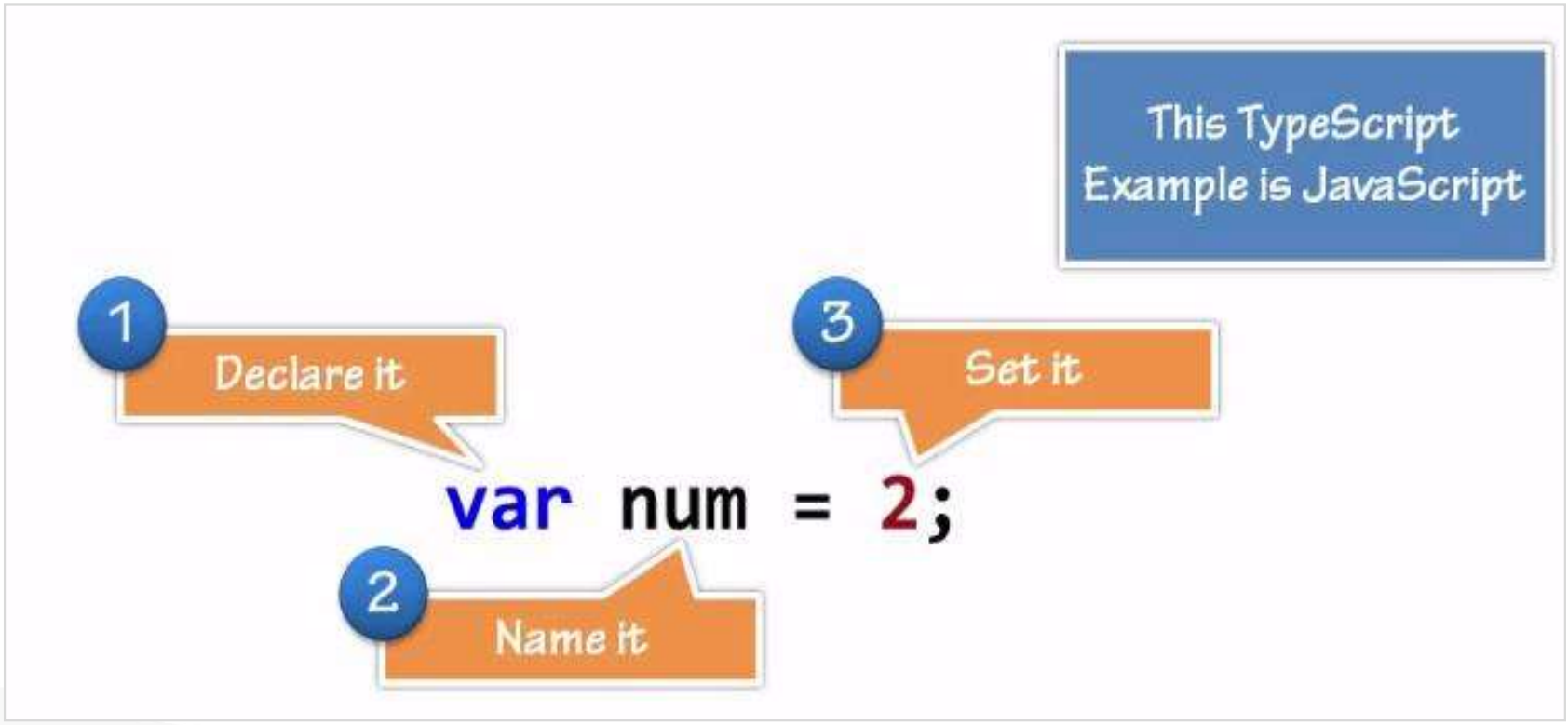
Keyword	Description
Modules	Encapsulation for code and classes
Imports	Import modules
Exports	Export a member from module
Class	Encapsulation for properties, variables and function members
Extends	Extend class. Used for inheritance
Implements	Implements an interface
Interface	Defines a contract of behaviour when implemented by types
Constructor	Provides initialization for classes
Public and Private	Member visibility modifier
=>	Arrow syntax, also called lambda expressions, used with definitions and functions
:	Separator between the variable name and the variable type
…	Reset syntax, the same as the C#'s methods parameter params
<TypeName>	Casting a type to another type
Generics<T>	Encapsulate operations that are not specific to a particular type
Enum	Used to quickly declare a range of constant or computed values

TypeScript Language Basics (3/8)

- TypeScript Code Hierarchy
  - In TypeScript, there is a very defined code hierarchy that is built-in Modules/Namespaces
  - Act as a naming container that can export different members.
  - One of the key things inside a module is a class.
  - Class is a container for Fields, Constructor, Properties and Functions.
  - A class can also implement interfaces for consistency across multiple classes.

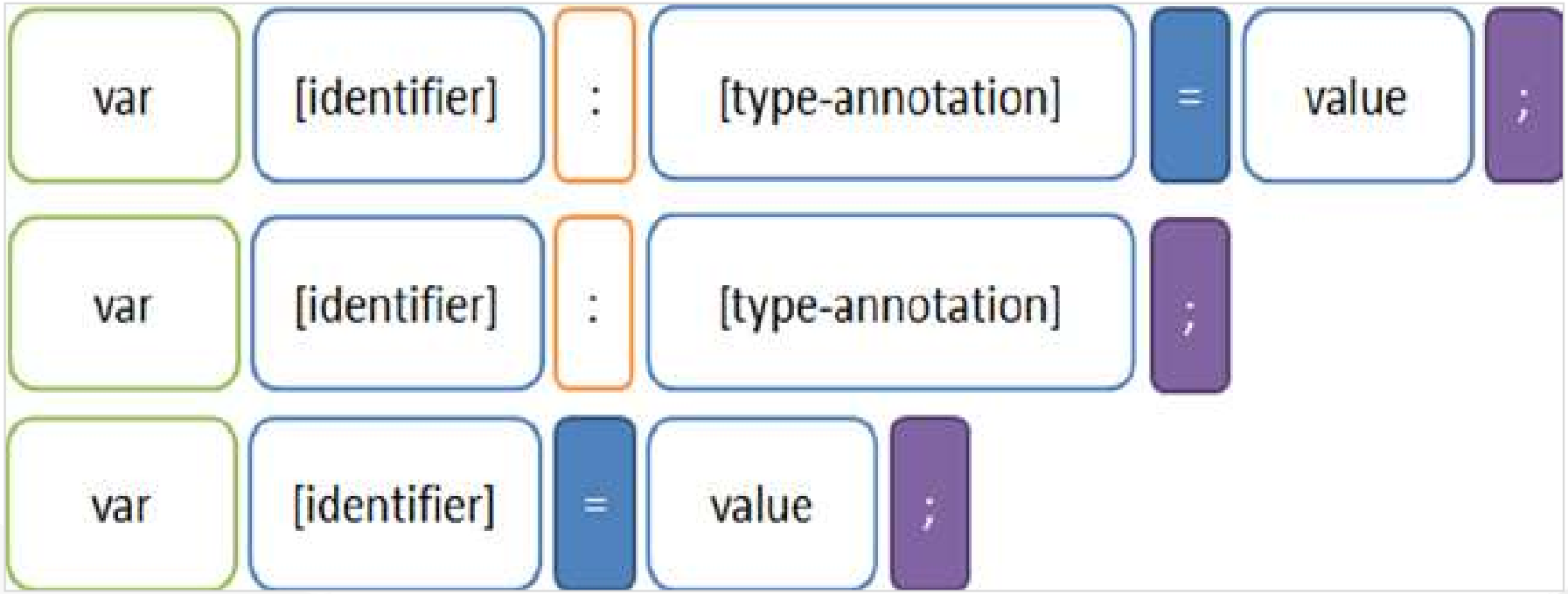
TypeScript Language Basics (4/8)

- TypeScript Type Inference
  - TypeScript language service is expert at inferring types automatically.



TypeScript Language Basics (5/8)

- At times, you may want to make a type explicit either for safety or readability. In all of these cases, you can use a type annotation to specify the type.
- For a variable, the type annotation comes after the identifier and is preceded by a colon.



TypeScript Language Basics (6/8)

- The type used to specify an annotation can be a primitive type, an array type, a function signature, or any complex structure you want to represent including the names of classes and interfaces you create.
- If you want to opt out of static type checking, you can use the special any type (base type of all other types in Typescript), which marks a variable's type as dynamic.
- No checks are made on dynamic types.

```
var any1;                                   Type could be any type (any)
var num1: number;                           Type Annotation
var num2: number = 2;                       Type Annotation Setting the Value
var num3 = 3;                               Type Inference (number)
var num4 = num3 + 100;                      Type Inference (number)
var str1 = num1 + 'some string';            Type Inference (string)
var nothappy : number = num1 + 'some string';   Error!
```

TypeScript Language Basics (7/8)

- More examples of type annotations:

```
//primitive type annotation
var name: string = 'Steve';
var heightInCentimeters: number = 182.88;
var isActive: boolean = true;

//array type annotation
var names: string[] = ['James', 'Nick', 'Rebecca', 'Lily'];

//function annotation with parameter type annotation and return type annotation
var sayHello: (name: string) => string;

//implementation of sayHello function
sayHello = function (name: string) {
   return 'Hello' + name;
};

//object type annotation
var person: { name: string; heightInCentimeters: number; }

//Implementation of a person object
person = {
   name: 'Mark',
   heightInCentimeters: 183
};
```

TypeScript Language Basics (8/8)

```
function Function1(arg): any
{
   return arg;
}
function Function2(): void
{
   return;
}
function Function3(arg: number)
{
   return arg;
}
var result1 = Function1(100);
var result2 = Function1('100');
var result3: any = Function1(true);
var result4: string = Function1('string');
var result5: boolean = Function1(true);
var result6 = Function3(100);
var result7: number = Function3(100);
```

TypeScript : Declaring Variables (1/2)

- ES2015 has introduced keywords let and const for declaring variables & TypeScript supports these.
- let is used to declare variables & const is used to declare constants.
- The primary difference between var and let & const is in the scoping rules applied.

var	let and const
Globally available in the function in which it is declared	Only available in the block in which it is declared
"Hoisted" to the top of the function	Not "hoisted" to the top of the block
Variable name may be declared a second time in the same function	Variable name may only be declared once per block

TypeScript : Declaring Variables (2/2)

- Example

```
function ScopeTest() {
    if (true){
        var foo = 'use anywhere';
        let bar = 'use in this block'
    }
    //do some more stuff
    console.log(foo); //works!!
    console.log(bar); //error!!
}
```

```
function F1()
{
    let x = 100;
    console.log("x before condition block is: " + x);
    if (10 < 20)
    {
        let x = 1000;
        console.log("x inside condition block is: " + x);
    }
    //let x = 100000; ERROR!!
}
F1();
```

TypeScript : Enumerations (1/2)

- Enumerations represent a collection of named elements that you can use to avoid littering your program with hard-coded values.
- They can be used to give friendly names to a specific set of numeric values'.
- By default, enumerations are zero based.
- This can be changed by specifying the first value, in which case numbers will increment from the specified value.

```
enum Category { Biography, Poetry, Fiction }; //0, 1, 2
enum Category { Biography = 1, Poetry, Fiction }; // 1, 2, 3
enum Category { Biography = 5, Poetry = 8, Fiction = 9 }; // 5, 8, 9

let favoriteCategory: Category = Category.Biography;

console.log(favoriteCategory); //5
let categoryString = Category[favoriteCategory]; // Biography
```

TypeScript : Enumerations (2/2)

```
enum Decisions {
    YES,
    NO = 20,
    UNDECIDED
}
var mydecision = Decisions.UNDECIDED;
if (mydecision == Decisions.UNDECIDED){
    alert("You have not yet decided?");
}
else {
    alert("Your decision is: " + Decisions[mydecision]);
}
```

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- **Functions**
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- References

TypeScript : Functions (1/10)

- Functions are the building blocks of readable, maintainable, and re-usable code.
- In TypeScript, most functions are actually written as methods that belong to a class.
- Functions are improved by a number of TypeScript language features.
- TypeScript Functions v/s JavaScript Functions:

TypeScript	JavaScript
Types (of course!)	No types
Arrow functions	Arrow functions (ES2015)
Function types	No function types
Required and optional parameters	All parameters are optional
Default parameters	Default parameters (ES2015)
Rest parameters	Rest parameters (ES2015)
Overloaded functions	No overloaded functions

TypeScript : Functions (2/10)

- Arrow Functions are also known as Lambda Functions.
- They provide a concise syntax for writing anonymous functions.

```
myBooks.forEach(() => console.log('Done reading!'));
myBooks.forEach(title => console.log(title));
myBooks.forEach((title, idx, arr) => console.log(idx + '-' + title));
myBooks.forEach((title, idx, arr) => {
    console.log(idx + '-' + title);
    //do more stuff here
});
```

TypeScript : Functions (3/10)

- Example

```
var AddNumbers = (a: number, b: number) => (a + b);
var result: number = AddNumbers(100, 200);
console.log(result);

var AddNumbersAgain = (a: number, b: number) =>
{
   console.log("AddNumbersAgain() called");
   return (a + b);
}
var v = AddNumbersAgain(1000, 2000);
console.log(v);
```

TypeScript : Functions (4/10)

- Sometimes the single expression to be returned by an arrow function will be an object.
- The braces around the object declaration confuses The TypeScript Compiler, so you need to mark it as an expression by surrounding it with parentheses.

```
var GetSomeObject = () => ({ Id: 1, Name: 'Karthik' });
var someObject = GetSomeObject();
console.log(someObject.Id + ' - ' + someObject.Name);
```

TypeScript : Functions (5/10)

- Capturing 'this' in JavaScript:
  In JavaScript, there is always a need to capture the value of 'this' before passing it to an anonymous function.

```
function Book()
{
    this.BookName = "TypeScript";
    var self = this;
    MainFunction(function ()
    {
        console.log(self.BookName);
    });
}
function MainFunction(callback)
{
    callback();
}
var b = new Book();
```

TypeScript : Functions (6/10)

- In arrow functions, you do not need to capture the value of ‘this’ into another variable.
- You can use it directly when using an arrow function.

```
function Book()
{
    this.BookName = "TypeScript";
    MainFunction(() =>
    {
        console.log(this.BookName);
    });
}
function MainFunction(callback)
{
    callback();
}
var b = new Book();
```

TypeScript : Functions (7/10)

- Optional Parameters
  - In JavaScript, all function parameters are optional.
  - In Typescript, by default all parameters are required.
  - To make a parameter optional, suffix the identifier with a question mark (?).
  - Optional parameters must be located after any required parameters in the parameter list.

TypeScript : Functions (8/10)

- Also, when you use an optional parameter you must check (inside the function) the value if it has been initialized.

```
function Add(n1: number, n2: number, n3?: number): number
{
   var result: number;

   if (n3 == undefined)
   {
      result = (n1 + n2);
      alert("Last parameter not specified");
   }
   else
   {
      result = (n1 + n2 + n3);
   }
   return result;
}
var ans = Add(10, 20, 30);
alert(ans);

ans = Add(10, 20);
alert(ans);
```

TypeScript : Functions (9/10)

- Default Parameters
  - Default parameters are complementary to optional parameters.
  - When you specify a default parameter, it allows the argument to be omitted by calling code and in cases where the argument is not passed the default value will be used instead.
  - Default value can be a literal or even an expression.
  - Default parameters behave as optional parameters when used at the end of all required parameters.
  - To supply a default value for a parameter, assign a value in the function declaration.

```
function Print(text: string, destination: string = 'Screen', fontsize: number): void
{
   console.log(text + " sent to destination: " + destination + " with fontsize: " + fontsize);
}

Print('HELLO', undefined, 20);
```

TypeScript : Functions (10/10)

- Rest Parameters
  - Rest parameters allow calling code to specify zero or more arguments of the specified type.
  - For the arguments to be correctly passed, rest parameters must follow these rules:
    - Only one rest parameter is allowed.
    - The rest parameter must appear last in the parameter list.
    - The type of a rest parameter must be an array type.
  - To declare a rest parameter, prefix the identifier with three periods (…) and ensure that the type annotation is an array type.

```
function DrawPolygon(point1: number, point2: number, ...remainingpoints: number[]): string
{
   var msg = "Point1 is: " + point1 + "\n";
   msg += "Point2 is: " + point2 + "\n";
   msg += "Total number remaining points are: " + remainingpoints.length;
   return msg;
}
alert(DrawPolygon(10, 20));
alert(DrawPolygon(10, 20, 30, 40, 50));
alert(DrawPolygon(10 ,20, 30));
```

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- Functions
- **Arrays & Tuples**
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- References

TypeScript : Arrays

- Arrays can be declared in two ways:

```
let strArray1: string[] = ['here', 'are', 'strings'];

let strArray2: Array<string> = ['here', 'are', 'strings'];
```

- Can be accessed and used much like JavaScript arrays.
- Declare an array to be of type any to store any type in the array.

```
let anyArray: any[] = [42, true, 'banana'];
```

TypeScript : Tuples

- Tuples are specialized arrays where types for the first few types are specified.
- Types need not be the same.
- New elements can be added to the Tuple as long as the types being stored in the element are one of the types declared previously.
- Primarily used as a base from which new types can be declared.

```
let mytuple: [number, string] = [25, 'abc'];
console.log(mytuple[0]); //prints 25
console.log(mytuple[1]); //prints 'abc'

mytuple.push('another string');//Works
console.log(mytuple[2]); //prints 'another string'

mytuple.push(100);//Works
console.log(mytuple[3]); //prints 100
//mytuple.push(true); //ERROR
```

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- **OOPS in TypeScript**
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- References

TypeScript : Interfaces (1/2)

- An interface can be used as an abstract type that can be implemented by concrete classes.
- It’s a contract that defines types.
  - Because interfaces define custom types & JavaScript does not use custom types, interfaces do not compile to anything in JavaScript. They are just used by the compiler for type checking.
- An interface specifies the shape of an object.
- Interfaces are also the building blocks for defining operations that are available in third-party libraries and frameworks that are not written
- Interfaces are declared with the interface keyword and contain a series of annotations to describe the contract that they represent.
- The annotations cannot only describe properties and functions, but also constructors and indexers.
- Interfaces are used at design time to provide auto completion and at compile time to provide type checking.

TypeScript : Interfaces (2/2)

```
interface Duck
{
    walk: () => void;
    swim: () => void;
    quack:() => void;
}

let probablyDuck = {
    walk: () => console.log("Walking like a duck"),
    swim: () => console.log("swimming like a duck"),
    quack: () => console.log("quacking like a duck")
};
function someFunction(bird: Duck) {
    bird.quack();
    bird.swim();
    bird.walk();
}
someFunction(probablyDuck);
```

- Interfaces for Function Types:
  - Functions have types which are defined by their parameters and return values.
  - This allows you to assign a function type to any variable.
  - To reuse that variable, we can assign it a name by defining an interface.

TypeScript : Classes (1/6)

- A class defines a template for creating an object, provide state storage and behaviour and Encapsulate reusable functionality.
- Classes in Typescript support:
  - Properties and Methods
  - Constructors
  - Access Modifiers
  - Inheritance
  - Abstract Classes
- Constructors
  - All classes in Typescript have a constructor, whether you specify or not.
  - If you leave out the constructor, the compiler will automatically add one.
  - For a class that doesn't inherit from another class, the automatic constructor will be parameterless and will initialize any class properties.

TypeScript : Classes (2/6)

- Where the class extends another class, the automatic constructor will match the superclass signature and will pass arguments to the superclass before initializing any of its own properties.
- Constructor parameters are not mapped to member variables. If you prefix a constructor parameter with an access modifier, such as private, it will automatically be mapped for you. You can refer to these constructor parameters as if they were declared as properties on the class.

```
class Employee
{
   constructor(private id: number, private name: string)
   {
      alert("Inside parameterized constructor");
   }
   GetDetails()
   {
      alert("EmpId: " + this.id + "\nEmpName: "+
this.name);
   }
}
var e = new Employee(2153, "KARTHIK");
e.GetDetails();
```

```
class Employee
{
   private EmpId: number;
   private EmpName: string;

   constructor(id: number, name: string)
   {
      this.EmpId = id;
      this.EmpName = name;
   }
   GetDetails()
   {
      alert("EmpId: " + this.EmpId + "\nEmpName: "+
this.EmpName);
   }
}
var e = new Employee(2153, "KARTHIK");
e.GetDetails();
```

TypeScript : Classes (3/6)

- Access Modifiers
  - Can be used to change the visibility of properties and methods within a class.
  - By default, properties and methods are public.
  - You need to prefix constructor parameters with the public keyword if you want them to be mapped to public properties automatically.
  - To hide a property or method, you prefix it with the private keyword. This restricts the visibility to within the class only, the member won't appear in auto completion lists outside of the class and any external access will result in a compiler error.
  - When you mark a class member as private, it can't even be seen by subclasses. If you need to access a property or method from a subclass, it must be made public.
  - No 'protected' modifier at present.
- Properties and Methods
  - Instance properties are typically declared before the constructor in a TypeScript class. A property definition consists of three parts; an optional access modifier, the identifier, and a type annotation.
  - You can also initialize the property with a value.
  - When your program is compiled, the property initializers are moved into the constructor. Instance properties can be accessed from within the class using 'this' keyword. If the property is public it can be accessed directly using the instance name.

TypeScript : Classes (4/6)

```
class Circle
{

   constructor(public Radius?: number)
   {
      if (Radius == undefined) {
         alert("Radius not passed while creating
circle");
      }
      else
      {
         alert("Circle created with radius: " +
this.Radius);
       }
   }
   CalculateArea(): number
   {
      return 3.14 * this.Radius * this.Radius;
   }
}

var c = new Circle();
c.Radius = 10.52;
var area = c.CalculateArea();
alert("Radius (from outside the class) is: " +
c.Radius);
alert("Area is : " + area);
```

```
class Circle
{
   //property explicitly defined
   public Radius: number;
   constructor(r?: number)
   {
      if (r == undefined) {
         alert("Radius not passed while creating circle");
      }
      else
      {
         this.Radius = r;
         alert("Circle created with radius: " + this.Radius);
       }
   }
   CalculateArea(): number
   {
      return 3.14 * this.Radius * this.Radius;
   }
}

var c = new Circle();
c.Radius = 10.52;
var area = c.CalculateArea();
alert("Radius (from outside the class) is: " + c.Radius);
alert("Area is : " + area);
```

TypeScript : Classes (5/6)

- Property setters & getters
  - TypeScript supports property getters and setters, as long as you are targeting ECMAScript 5 or above.
  - The syntax for these is identical to method signatures as described in the following, except they are prefixed by either the get or set keyword.

```
class Book
{
   private bookid: number; private bookname: string;
   get BookCode()
   {
      alert("Getting Bookcode......"); return this.bookid;
   }
   set BookCode(bc: number)
   {
      alert("Setting BookCode......"); this.bookid = bc;
   }

   get BookName()
   {
      alert("Getting Bookname......."); return this.bookname;
   }
   set BookName(bn: string)
   {
      alert("Setting Book..........."); this.bookname = bn;
   } }

var b = new Book();
b.BookCode = 100;
b.BookName = "Angular JS";

alert("Book code: " + b.BookCode);
alert("Book Name: " + b.BookName);
```

TypeScript : Class Heritage (6/6)

- There are two types of class heritage in TypeScript.
  - A class can implement an interface using the implements keyword.
  - A class can inherit from another class using the extends keyword.
- A class can implement multiple interfaces, with each interface being separated by a comma.

```
 interface Greeter
{
   Greet(): string;
}

class Morning implements Greeter
{
   public name: string;
   Greet(): string
   {
      return "GOOD MORNING!!" + this.name;
   }
}

class Evening implements Greeter
{
   public name: string;
   Greet(): string
   {
      return "GOOD EVENING!!" + this.name;
   }
}

var m = new Morning(); m.name = "KARTHIK";
alert(m.Greet());

var e = new Evening(); e.name = "AJIT";
alert(e.Greet());
```

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- **Modules in TypeScript**
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- References

Difference between Internal and External Modules

- Internal Modules
  - Internal modules are local or exported members of other modules.
  - In case of internal modules, when you compile your typescript files your modules are converted into variables that nest as necessary to form namespace-like objects.
  - Internal modules are declared using ModuleDeclarations that specify their name and body.
  - A name path with more than one identifier is equivalent to a series of nested internal module declarations.
- External Modules
  - External modules are separately loaded bodies of code referenced using external module names.
  - An external module is written as a separate source file that contains at least one import or export declaration.
  - In addition, external modules can be declared using AmbientModuleDeclarations in the global module that directly specify the external module names as string literals.
  - External modules are useful in sense they hide the internal statements of the module definitions and show only the methods and parameters associated to the declared variable.

External Modules in TypeScript (1/3)

- External modules help us manage large amount of dependencies in large JavaScript-based applications.
- Benefits of Internal Modules:
  - Provide a namespace-like syntax.
  - Good for grouping common sets of functionality.
  - Ideal for smaller projects & no need to import them into another module.
- Benefits of External Modules
  - Separately loadable modules which can be loaded separate from each other & only when needed.
  - Exported entities in an external module can be imported into another module.

```
import viewmodels = require('./viewmodels')
```

External Modules in TypeScript (2/3)

- Typescript uses AMD and CommonJS conventions to load external modules.
- Why External Modules?
  - Sequencing script dependencies is very difficult in large applications.
- AMD (Asynchronous Module Definition)
  - Helps loading modules asynchronously as and when required.
  - Manages module dependencies.
  - Loads modules in a sequence based on who requires which modules.
  - Use the keyword ‘require’ to specify the dependencies.
- To use AMD with external modules, we need require.js to manage these dependencies.

External Modules in TypeScript (3/3)

```
 require(['bootstrapper'],
    (bootstrapper) => {
        bootstrapper.run();
    });

import gt = namespace('./greeter');
export function run() {
    var el = document.getElementById('content');
    var greeter = new gt.Greeter(el);
    greeter.start();
}

export class Greeter {
    start() {
        this.timerToken = setInterval(() =>
        this.span.innetText = new Date().toUTCString(), 500);
    }
}
```

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- **Type Definitions**
- Generics & Enumerations
- Unions & Type Alises
- References

Type Definitions (1/2)

- Type Definitions allow you to incorporate JavaScript libraries into your TypeScript projects.
- DefinatelyTyped is a huge repository of Type Definition files for any JavaScript library.
- Type Definitions files can be directly downloaded from the DefinatelyTyped GitHub repository
- There are tools for obtaining and managing them
- What are Type Definition Files?
  - These are files that can be added to a project that contain type information for a library intended to be used.
  - They contain no implementation details & mainly consist of interfaces that make it possible for The TypeScript Compiler to type-check the code against the library being used.

Type Definitions (2/2)

- Mostly these files are used as a TypeScript wrapper for JavaScript libraries and can be used for TypeScript code as well.
- They provide structure and strong type-checking for lots of popular JavaScript libraries and frameworks.
- Primary benefit is for Design-Time tool for type-checking and editor support.
- Files end with a .d.ts extension.
- The DefinatelyTyped Repository
  - Type definitions repository can be downloaded in several ways.
    - Direct download from GitHub. (https://github.com/DefinitelyTyped/DefinitelyTyped)
    - Nuget (in case you are using Visual Studio)
    - tsd (a Type Definition manager developed specifically to work with DefinatelyTyped)
    - typings (a Type Definition manager capable of working with multiple sources, DefinatelyTyped being one of them)

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- **Generics & Enumerations**
- Unions & Type Alises
- References

Generics (1/3)

- Generics allow creating a component that can work over a variety of types rather than a single one
  - This allows users to consume these components and use their own types
- Without generics, we would either have to give the identity function a specific type:

```
function identity(arg: number): number { return arg; }
```

- OR
- We could describe the identity function using the any type

```
function identity(arg: any): any { return arg; }
```

- In the second case, we actually are losing the information about what that type was when the function returns
  - Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned

Generics (2/3)

- Use a type variable, a special kind of variable that works on types rather than values

```
function identity<T>(arg: T): T { return arg; }
```

- **T** allows us to capture the type the user provides (e.g. number), so that we can use that information later
  - This version of the identity function is generic, as it works over a range of types
- We can call a generic method in one of two ways:

```
let output = identity<string>("myString"); // type of output will be 'string'
```

```
let output = identity("myString"); // type of output will be 'string'
```

Generics (3/3)

- Generic classes:
  - Generic classes have a generic type parameter list in angle brackets (<>) following the name of the class

```
class GenericNumber<T> { zeroValue: T; add: (x: T, y: T) => T; }
let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

```
let stringNumeric = new GenericNumber<string>(); stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };
console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Enumerations (1/2)

- Allow us to define a set of **named constants**
  - Makes it easier to document intent, or create a set of distinct cases
  - TypeScript provides both numeric and string-based enums
- **Numeric enums**

```
enum Direction { Up = 1, Down, Left, Right, }
```

- All of the following members are auto-incremented from that point on.
  - Direction.Up has the value 1, Down has 2, Left has 3, and Right has 4
  - Initializers can be left off as well

```
enum Direction { Up, Down, Left, Right, }
```

- Up would have the value 0, Down would have 1, etc

Enumerations (2/2)

- Any member can be accessed as a property off of the enum itself, and types can be declared using the name of the enum

```
enum Response { No = 0, Yes = 1, }
function respond(recipient: string, messageca: Response): void { // ... }
respond("Princess Caroline", Response.Yes)
```

- **String enums**
  - In a string enum, each member has to be constant-initialized with a string literal, or with another string enum member

```
enum Direction { Up = "UP", Down = "DOWN", Left = "LEFT", Right = "RIGHT", }
```

  - While string enums don't have auto-incrementing behavior, string enums have the benefit that they **serialize** well
  - String enums allows giving a meaningful and readable value when the code runs, independent of the name of the enum member itself

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- **Unions & Type Alises**
- References

Unions (1/3)

- What if there is a library that expects a parameter to be either a number or a string?

```
/** * Takes a string and adds "padding" to the left. * If 'padding' is a string, then 'padding' is appended to the left side. * If 'padding' is a number, then that number of spaces is added to the left side. */
function padLeft(value: string, padding: any)
{
        if (typeof padding === "number")
        {
                return Array(padding + 1).join(" ") + value;
        }
        if (typeof padding === "string")
        {
                return padding + value;
        }
        throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns " Hello world"
```

Unions (2/3)

- Since **padLeft()** function's **padding** parameter is typed as **any**, we can call it with an argument that's neither a number nor a string, but TypeScript will be okay with it
- Instead of **any**, we can use a union type for the padding parameter

```
/** * Takes a string and adds "padding" to the left. * If 'padding' is a string, then 'padding' is appended to the left side. * If 'padding' is a number, then that number of spaces is added to the left side. */

function padLeft(value: string, padding: string | number)
{ // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```

- A union type describes a value that can be one of several types. We use the vertical bar (|) to separate each type, so number | string | boolean is the type of a value that can be a number, a string, or a boolean

Unions (3/3)

- If we have a value that has a union type, we can only access members that are common to all types in the union

```
interface Bird { fly(); layEggs(); }

interface Fish { swim(); layEggs(); }

function getSmallPet(): Fish | Bird { // ... }

let pet = getSmallPet();
pet.layEggs(); // okay
pet.swim(); // errors
```

Type Alises (1/2)

- Type aliases create a new name for a type
  - Type aliases are sometimes similar to interfaces, but can name primitives, unions, tuples, and any other types that otherwise had to be written by hand

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;

function getName(n: NameOrResolver): Name
{
        if (typeof n === "string")
        {
                return n;
        }
        else
        {
                return n();
        }
}
```

Type Alises (2/2)

- Aliasing doesn't actually create a new type - it creates a **new name** to refer to that type
- Just like interfaces, type aliases can also be generic - we can just add type parameters and use them on the right side of the alias declaration

```
type Container<T> = { value: T };
```

- A type alias can also refer to itself in a property

```
type Tree<T> = { value: T; left: Tree<T>; right: Tree<T>; }
```

Agenda

- Introduction to TypeScript
- TypeScript Compiler
- Installing TypeScript
- TypeScript Language Basics
- Functions
- Arrays & Tuples
- OOPS in TypeScript
- Modules in TypeScript
- Type Definitions
- Generics & Enumerations
- Unions & Type Alises
- **References**

References

- https://www.Typescriptlang.org/
- http://www.datchley.name/es6-promises/
- http://www.datchley.name/promise-patterns-anti-patterns/
- http://slides.com/hooliganlin/ng-promises#/8
- http://bahmutov.calepin.co/why-promises-need-to-be-done.html
- http://www.jeremyzerr.com/sites/default/files/AngularJS-Promises.pdf

Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email Ct-univerct@citiustech.com

www.citiustech.com