
TS
TypeScript

Introduction

- TypeScript is a language designed by Anders Hejlsberg (designer of C#) at Microsoft and released under an open-source Apache 2.0 License (2004)
- Typed superset of JavaScript
- Transpiled to JavaScript (source-to-source compilation)
- Supports ECMAScript5, ECMAScript6 & additional features (generics, type annotations)
- Pure object oriented with classes, interfaces and statically typed
- The popular JavaScript framework Angular 2.0 is written in TypeScript

Why TypeScript?

- As JavaScript code grows, it tends to get messier, making it difficult to maintain and reuse the code
- Design-time tools, compile-time checking, dynamic module loading at runtime
- Highly portable (web browser, web server, web application, native application on OS)

Installation

- Download and install Node.js from *https://nodejs.org/en/download*

- Use NPM commands to install TypeScript globally:

```
npm install -g typescript
```

First TypeScript Code

hello.ts

```
var message : string = "Hello World!";

console.log(message);
```

To compile typescript file: tsc hello.ts

To run typescript file: node hello.js

ts-node package

- npm install –g ts-node

- To compile and run typescript file:

```
ts-node   hello.ts
```

Identifiers

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

Keywords

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

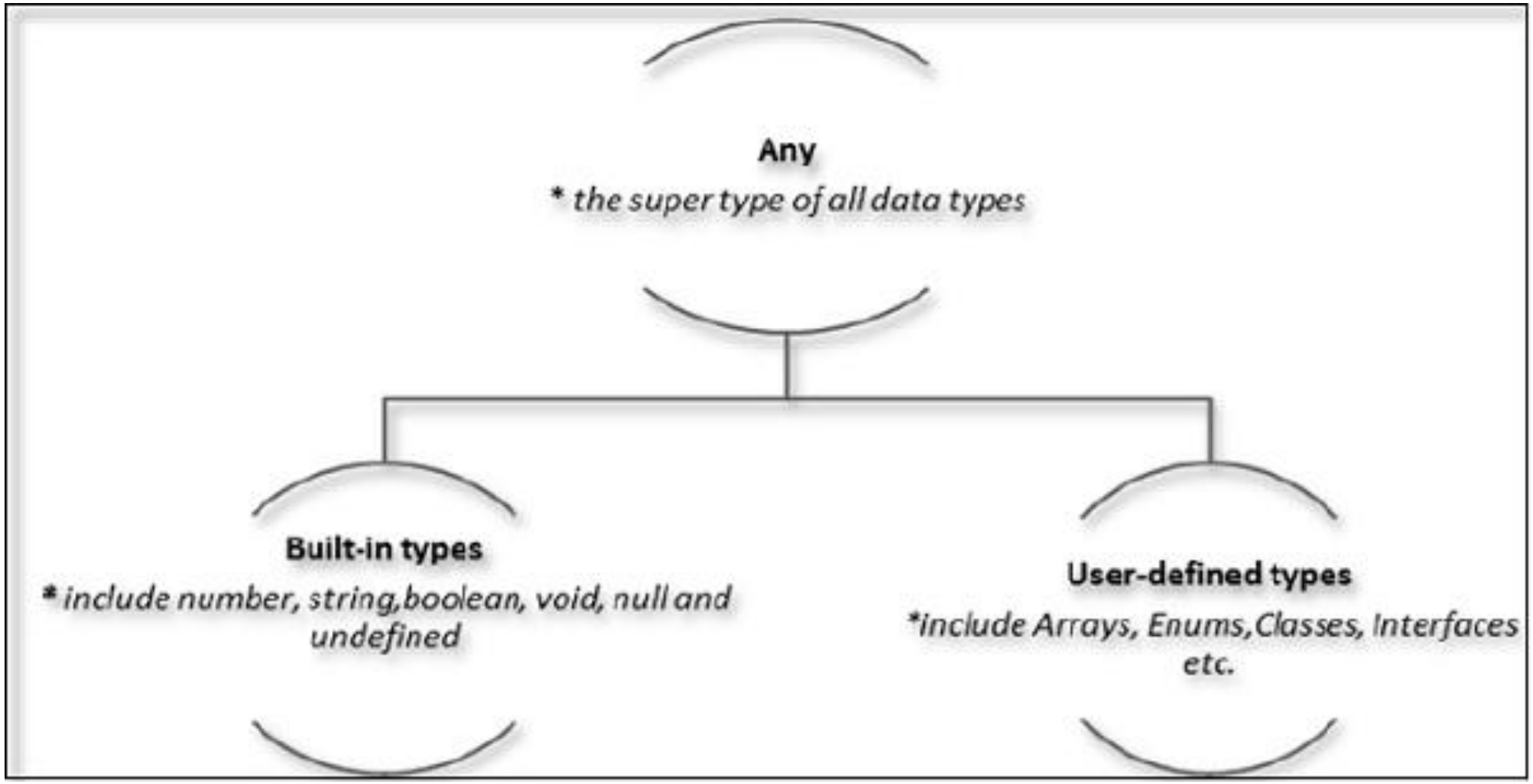
Objects (greet.ts)

```
class Greeting
{
    greet():void {
        console.log("Hello World!!!")
    }
}

var obj = new Greeting();

obj.greet();
```

Objects (greet.js)

```
var Greeting = /** @class */ (function () {
    function Greeting() {
    }
    Greeting.prototype.greet = function () {
        console.log("Hello World!!!");
    };
    return Greeting;
}());

var obj = new Greeting();

obj.greet();
```

Types


Any
* the super type of all data types
Built-in types
* include number, string,boolean, void, null and undefined
User-defined types
*include Arrays, Enums,Classes, Interfaces etc.

Built-in Types

Data type	Description
number	Double precision 64-bit floating point values.
string	Represents a sequence of Unicode characters
boolean	Represents logical values, true and false
void	Used on function return types to represent non-returning functions
null	Represents an intentional absence of an object value.
undefined	Denotes value given to all uninitialized variables

Variable declaration

var name:string = "mary"
var name:string;
var name = "mary"
var name;

Type Assertion

```
let message;
```

type **any**

```
message = 'How are you?';

let res = (<string>message).includes(message);
```

OR

```
let res = (message as string).includes(message);

console.log(res);
```

Type Inference

```
var num = 2;

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

num = "12";
//Type 'string' is not assignable to   type 'number'

console.Log(num);
```

Variable Scope

- **Global Scope**– declared outside the programming constructs. Can be accessed from anywhere within your code.
- **Class Scope (Fields)** – declared within the class but outside the methods. Can be accessed using the object of the class. Fields can also be static. Static fields can be accessed using the class name.
- **Local Scope** – declared within the constructs like methods, loops etc. Are accessible only within the construct where they are declared.

Variable Scope (Ex.)

```
var global_num = 12
class Numbers {
    num_val = 13;
    static sval = 10;
    storeNum(): void {
        var local_num = 14;
    }
}
console.log("Global num: " + global_num);
console.log(Numbers.sval);
var obj = new Numbers();
console.log("Global num: " + obj.num_val)
```

Functions with optional parameter

```
function disp(id: number,
              name: string,
              mail_id?: string)
{
    console.log("ID:", id);
    console.log("Name", name);

    if (mail_id != undefined)
        console.log("Email Id", mail_id);
}
disp(123, "John");
disp(111, "mary", "mary@xyz.com");
```

Rest Parameters

```
function addNumbers(...nums: number[]) {
    var i;
    var sum: number = 0;
    for (i = 0; I < nums.length; i++) {
        sum = sum + nums[i];
    }
    console.log("sum : ", sum)
}
addNumbers(1, 2, 3)
addNumbers(10, 10, 10, 10, 10)
```

Default Parameters

```
function calc(price: number,
              rate: number = 0.50)
{
    var discount = price * rate;
    console.log("Discount Amount: ", discount);
}
calc(1000)
calc(1000, 0.30)
```

Lambda/Arrow Functions

```
var plus10 = (x: number) => 10 + x

console.log(plus10(100))
//outputs 110
```

Parameter Type Inference

```
var func = (x) => {
    if (typeof x === "number") {
        console.log(x + " is numeric")
    } else if (typeof x === "string") {
        console.log(x + " is a string")
    }
}
func(12)
func("Tom")
```

Function Overloads

Step – 1: Declare multiple functions with different signatures

Step – 2: Define the function. Give type any if parameter type differs. Mark parameters optional if no. of parameters are different

Step – 3: Invoke the function

Function Overloads (Ex.)

```
function disp(s1: string): void;

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

function disp(x: any, y?: any): void {
    console.log(x);
    if (y != undefined)
        console.log(y);
}
disp("abc")
disp(1, "xyz");
```

Arrays

```
var alphas: string[] = ["a", "b", "c", "d"]
console.log(alphas[0], alphas[1]);

var nums: number[] = [1, 2, 3, 4]
console.log(nums[0], nums[1]);
```

An Array Object (Ex.1)

```
var arr_names: number[] = new Array(4)

for (var i = 0; i < arr_names.length; i++)
{
    arr_names[i] = i * 2
    console.log(arr_names[i])
}
```

An Array Object (Ex.2)

```
var names: string[] = new Array("Mary", "Tom", "Jack")

for (var i = 0; i < names.length; i++)
{
    console.log(names[i])
}
```

Narrowing

- The process of refining types to more specific types than declared is called *narrowing*

```
function padLeft(padding: number | string, input: string)
{
    if (typeof padding === "number")
    {
        return new Array(padding + 1).join(" ") + input;
    }
    return padding + input;
}
```

```
(parameter) padding: number
```

```
(parameter) padding: string
```

Tuples

- Represents a heterogeneous collection of values
- Enable storing multiple fields of different types
- Can be passed as parameters to functions

Tuples

```
var mytuple = [10,"Hello"];

console.log(mytuple[0])
console.log(mytuple[1])

var tup = []
tup[0] = 12
tup[1] = "hi"

console.log(tup[0])
console.log(tup[1])
```

Unions

```
var val: string | number

val = 12
console.log("numeric value of val " + val)

val = "This is a string"
console.log("string value of val " + val)
```

Union type and Function parameter

```
function disp(name: string | string[]) {
    if (typeof name == "string") {
        console.log(name)
    } else if (name instanceof Array) {
        var i;
        for (i = 0; i < name.length; i++) {
            console.log(name[i])
        }
    }
}
disp("mark")
console.log("Printing names array....")
disp(["Mark", "Tom", "Mary", "John"])
```

Interfaces

- An interface is a syntactical contract that an entity should conform to.
- Interfaces define properties, methods, and events, which are the members of the interface.
- Interfaces contain only the declaration of the members.
- It is the responsibility of the deriving class to define the members.

Why Interface?

```
function disp_cust(customer: any)
{
   console.log("Customer Object: ")
   console.log(customer.firstName)
   console.log(customer.lastName)
}
disp_cust({ fName: "John", lName: "Doe" })

//Customer Object:
//undefined
//undefined
```

Interface (Ex)

```
interface IPerson {
    firstName: string,
    lastName: string,
    sayHi: () => string
}
var customer: IPerson =
{
    firstName: "Tom",
    lastName: "Hanks",
    sayHi: (): string => {
        return "Hi there"
    }
}
```

```
function disp_cust2(
                customer: IPerson)
{
    console.log("Customer Object: ")
    console.log(customer.firstName)
    console.log(customer.lastName)
    console.log(customer.sayHi())
}
disp_cust2(customer)
```

Union and Interfaces

```
interface RunOptions {
    program: string;
    cmdl: string[] | string | (() => string);
}

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

console.log(options.cmdl)
```

Union and Interfaces

```
options = {
      program: "test1",
      cmdl: ["Hi", "Brother"]
};

console.log(options.program);
console.log(options.cmdl);
```

Union and Interfaces

```
options = {
    program: "test1",
    cmdl: () => {
        return "Hi Everyone!";
    }
};

var fn: any = options.cmdl;

console.log(fn());
```

Interfaces and Arrays

```
interface namelist {
    [index: number]: string
}
var list2: namelist = ["John", 1, "Bran"] //Error
```

Interface Inheritance

```
interface Person {
    age: number
}
interface Musician extends Person {
    instrument: string
}
var drummer = <Musician>{};
drummer.age = 27
drummer.instrument = "Drums"

console.log("Age: "          + drummer.age)
console.log("Instrument: "   + drummer.instrument)
```

Multiple Interface Inheritance

```
interface IParent1 {
    eye_color: string
}
interface IParent2 {
    hair_color: string
}
interface Child extends IParent1, IParent2 {
}
var obj: Child = {
     eye_color: "brown", hair_color: "black"
}
console.log(obj.eye_color + " " + obj.hair_color)
```

Class

```
class Car {
    engine: string;   //field

    constructor(engine: string) {
        this.engine = engine
    }
    disp(): void {
        console.log("Engine is : " + this.engine)
    }
}
var obj = new Car("Engine1");
```

Class Inheritance

```
class Shape {
    area: number
    constructor(area: number) {
        this.area = area
    }
}
class Circle extends Shape {
    disp(): void {
        console.log("Area of the circle: " + this.area)
    }
}
var obj = new Circle(223);
obj.disp()
```

Method Overriding

```
class PrinterClass {
    doPrint(): void {
        console.log("doPrint() from Parent called…")
    }
}
class StringPrinter extends PrinterClass {
    doPrint(): void {
        super.doPrint()
        console.log("doPrint() is printing a string…")
    }
}
var obj = new StringPrinter()
obj.doPrint()
```

The static keyword

```
class StaticMem {
    static num: number;

    static disp(): void {
        console.log("num=" + StaticMem.num)
    }
}
StaticMem.num = 12
StaticMem.disp()
```

The instanceof operator

```
class Person {
}
var obj = new Person()

var isPerson = obj instanceof Person

console.log("obj is an instance of Person" + isPerson)
// true
```

Access Modifiers (public - default)

```
class Animal {
    public name: string;
    public constructor(name: string)
    {
        this.name = name;
    }
    public move(meter: number)
    {
        console.log("this.name" + "moved" + meter);
    }
}
```

Access Modifiers (private)

```
class Animal {
    private name: string;

    constructor(name: string) {
        this.name = name;
    }
}
new Animal("Cat").name;     // Error
```

Access Modifiers (protected)

```
class Person {
    protected name: string;
    constructor(name: string) { this.name = name; }
}
class Employee extends Person {
    private department: string;
    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    } public getIntro() {
        return this.name + this.department;
    }
}
let john = new Employee("John", "Sales");
console.log(john.getIntro());
console.log(john.name); // Error
```

Access Modifiers (readonly)

```
class Person {
     readonly pan_no: string;
      constructor(pan_no: string)
      {
          this.pan_no = pan_no;
      }
      public getPanNo() {
          return this.pan_no;
      }
}
let john = new Person("AEPRS1234L");
john.pan_no = "AEMTQ2345S";                    // Error
```

Parameter Properties

```
class Person {
    private _name: string;
    private _age: number;
    constructor(name: string, age: number) {
        this._name = name;
        this._age = age;
    }
}
```

```
class Person {
    constructor(private _name: String,
                private _age: number) {
    }
}
```

Classes and Interfaces

```
interface ILoan {
    interest: number
}
class AgriLoan implements ILoan {
    interest: number
    rebate: number
    constructor(interest: number, rebate: number) {
        this.interest = interest
        this.rebate = rebate
    }
}
var obj = new AgriLoan(10, 1)
console.log(obj.interest + " : " + obj.rebate)
```

Type Template

In JS you can add a method to an object literal after creation of an object.

```
var person = {
    firstName: "Tom",
    lastName: "Hanks"
}
person.sayHello = function () {
    return "hello"
}
```

This is not allowed in TS, because in TS concrete objects should have a type template.

Type Template

```
var person = {
    firstName: "Tom",
    lastName: "Hanks",
    sayHello: function () { }  //Type template
}
person.sayHello = function () {
    console.log("hello " + this.firstName)
}
person.sayHello();
```

Objects as function parameters

```
var person = {
    fname: "Tom",
    lname: "Hanks"
};

var disp = function (obj: {fname:string, lname:string})
{
    console.log("first name :" + obj.fname)
    console.log("last name :"  + obj.lname)
}

disp(person)
```

Inline Annotation

Anonymous Object

```
var disp = function (obj: {fname:string, lname:string})
{
    console.log("first name :" + obj.fname)
    console.log("last name :"  + obj.lname)
}

disp({ fname: "Tom", lname: "Hanks" })
```

Type Aliases

- If we want to use the same type more than once and refer to it by a single name.

```
type Point = {
  x: number;
  y: number;
};

function printCoord(pt: Point) {
  console.log(pt.x ", " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

```
type ID = number | string
```

Type Aliases vs Interfaces

```
// Adding new fields to an exist
ing interface

interface Window {
    title: string
}

interface Window {
    ts: TypeScriptAPI
}
```

```
// A type cannot be changed
after being created

type Window = {
    title: string
}

type Window = {
    ts: TypeScriptAPI
}

// Error: Duplicate identifier
 'Window'.
```

Type Aliases vs Interfaces

```ts
// Extending an Interface

interface Animal {
    name: string
}

interface Bear extends Animal {
    honey: boolean
}
```

```ts
//Extending via Intersections

type Animal = {
    name: string
}

type Bear = Animal & {
    honey: boolean
}
```

Duck Typing

- Two objects are considered to be of the same type if both share the same set of properties

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

Duck Typing (Ex.)

```
interface IPoint {
    x: number
    y: number
}
function addPoints(p1: IPoint, p2: IPoint): IPoint {
    var x_cord = p1.x + p2.x
    var y_cord = p1.y + p2.y
    return { x: x_cord, y: y_cord }
}
//Valid
var newPoint = addPoints({ x: 3, y: 4 }, { x: 5, y: 1 })

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

Function Type Expressions

A function with one parameter, named **a**, of type string, that doesn't have a return value

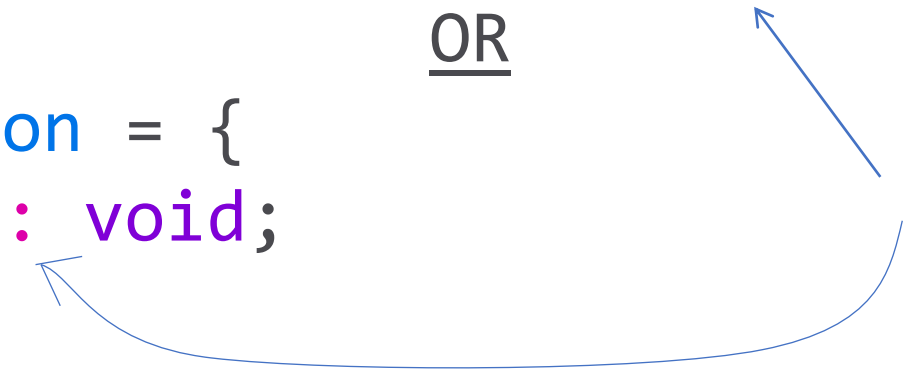
```
function greeter(fn: (a: string) => void) {
    fn("Hello, World");
}

function printToConsole(s: string) {
    console.log(s);
}

greeter(printToConsole);
```

Function Signature (Ex.1)

```
type GreetFunction = (a: string) => void;
```

OR

```
type GreetFunction = {
    (a: string) : void;
}
```




```
function greeter(fn: GreetFunction) {
  // ...
}
```

Note:- The parameter name is **required**. The function type **(string) => void** means “a function with a parameter named string of type any”!

Function Signature (Ex.2)

```
type add = (a:number, b:number) => number

let addNow:add = (a, b) => {
    return a+b
}
```

Function Signature (Ex.3)

```
function double(
    fn: (index: number) => void,
    n: number
){
    for (let i = 0; i < n; i++) {
        fn(i * 2);
    }
}

double(n => console.log(n), 4)
// 0 2 4 6
// Of course on a separate line!
```

Namespace

- In JavaScript variable declarations go into a global scope.
- If multiple JS files are used within same project there will be possibility of overwriting or mis-constructing the same variable.
- Which will lead to the "global namespace pollution problem".
- The namespace is used for logical grouping of functionalities.
- A namespace can include interfaces, classes, functions and variables.
- Earlier terminology was to refer it as “internal modules”.

Namespace (StringUtility.ts)

```
namespace StringUtility
{
	export function ToCapital(str: string): string
	{
		return str.toUpperCase();
	}
	export function SubString( str: string,
				   from: number,
				   length: number = 0): string
	{
		return str.substr(from, length);
	}
}
```

Namespace (Employee.ts)

```
/// <reference path="StringUtility.ts" />
class Employee {
    empCode: number;
    empName: string;
    constructor(code: number, name: string) {
        this.empCode = code;
        this.empName = StringUtility.ToCapital(name);
    }
    displayEmployee() {
        console.log(this.empCode + ", " + this.empName);
    }
}
const emp = new Employee(5, "John");
emp.displayEmployee();
```

Compiling and Executing Namespaces

```
> tsc  --target  es6   .\Employee.ts   --outfile   .\Employee.js

> node   .\Employee.js
```

To compile multiple namespaces into a single .js file

```
> tsc  --outFile  File.js  File1.ts  File2.ts  File3.ts
```

Nested Namespaces

```
namespace FirstLevel {
    export namespace SecondLevel {
        export class Example {
        }
    }
}

const nested = new FirstLevel.SecondLevel.Example();
```

Dotted Namespaces

```
namespace FirstLevel.SecondLevel.ThirdLevel {
    export class Example {
    }
}

const dotted =
    new FirstLevel.SecondLevel.ThirdLevel.Example();
```

Modules

- A module is used to organize code.
- A module is a way to create a group of related variables, functions, classes, and interfaces, etc.
- It executes in the **local scope**, not in the **global scope**.
- We can create a module by using the **export** keyword and can use in other modules by using the **import** keyword.
- Just like namespaces, modules can contain both code and declarations. The main difference is that modules *declare* their dependencies.

Modules

- Modules import another module by using a **module loader**.
- At runtime, the module loader is responsible for locating and executing all dependencies of a module before executing it.
- The most common modules loaders which are used in JavaScript are
  1) the **CommonJS** module loader for Node.js and
  2) **require.js** for Web applications.
- We can divide the module into **two** categories:
  1) Internal Module
  2) External Module

Internal Modules

- Internal modules were in the earlier version of Typescript.
- It was used for logical grouping of the classes, interfaces, functions, variables into a single unit and can be exported in another module.
- The modules are named as a namespace in the latest version of TypeScript.
- Today, internal modules are obsolete.
- But they are still supported by using namespace over internal modules.

External Modules

- External modules are also known as a module.
- External Module is used to specify the load dependencies between the multiple external js files.
- In TypeScript, just as in ECMAScript 2015, any file containing a top-level *import* or *export* is considered a module.
- A file without any top-level import or export declarations is treated as a script whose contents are available in the global scope.

ES Module (Ex.1)

hello.ts

```ts
export default function helloWorld() {
    console.log("Hello, world!");
}
```

import_hello.ts

```ts
import hello from "./hello.js"

hello();
```

```
> tsc  .\hello.ts .\import_hello.ts
> node  .\import_hello.js
```

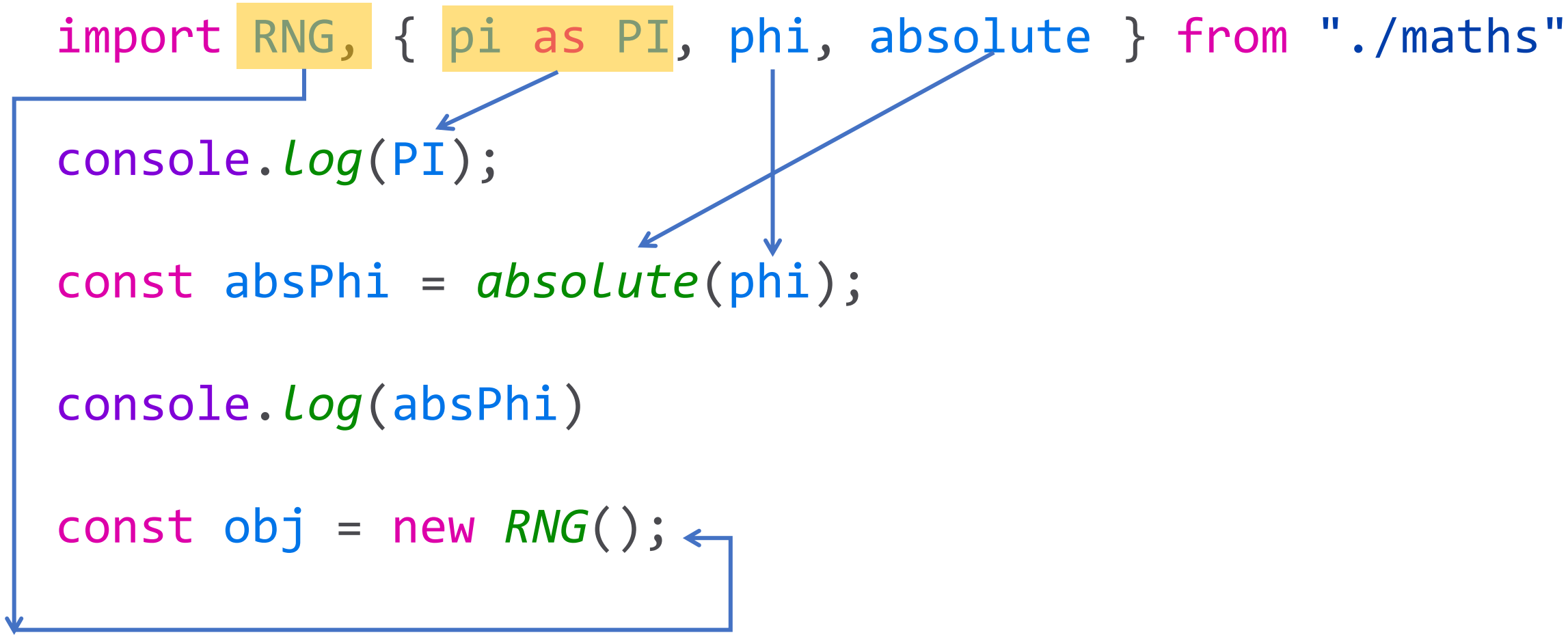
ES Module (Ex.2)

maths.ts

```
export var pi = 3.14;
export let squareTwo = 1.41;
export const phi = 1.61;

export default class RandomNumberGenerator { }

export function absolute(num: number) {
    if (num < 0) return num * -1;
    return num;
}
```

ES Module (Ex.2 ..cont)

import_maths.ts



ES Module (Ex.3)

- You can take all of the exported objects and put them into a single namespace using * as name:

```
import * as math from "./maths.js";

console.log(math.pi);

const positivePhi = math.absolute(math.phi);
```

CommonJS Module

- CommonJS is the format which most modules on npm are delivered in.
- Identifiers are exported via setting the exports property on a global called module.

CommonJS Module (Ex.1)

commonjs.ts

```
function absolute(num: number) {
    if (num < 0)
        return num * -1;
    return num;
}

module.exports = {
    pi: 3.14,
    squareTwo: 1.41,
    phi: 1.61,
    absolute,
};
```

CommonJS (Ex.1)

```
const maths = require("./commonjs.js")

console.log(maths.pi)
```

OR

```
const { pi } = require("./commonjs.js");

console.log(pi)
```

References

- https://www.typescriptlang.org/docs/
- https://www.tutorialspoint.com/typescript/index.html
- https://www.javatpoint.com/
- "Pro Typescript" by Steve Fenton, 2nd Edition, Apress, 2018