



# TypeScript Fundamentals

---

.NET CORE

*The goal of TypeScript is to be a static typechecker for JavaScript programs - in other words, a tool that runs before your code runs (static) to ensure that the **types** of the program are correct (typechecked).*

[HTTPS://WWW.TYPESCRIPTLANG.ORG/DOCS/HANDBOOK/INTRO.HTML#ABOUT-THIS-HANDBOOK](https://www.typescriptlang.org/docs/handbook/intro.html#about-this-handbook)

# TypeScript – Overview

<https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html#typescript-a-static-type-checker>  
<https://angular.io/guide/glossary>

---

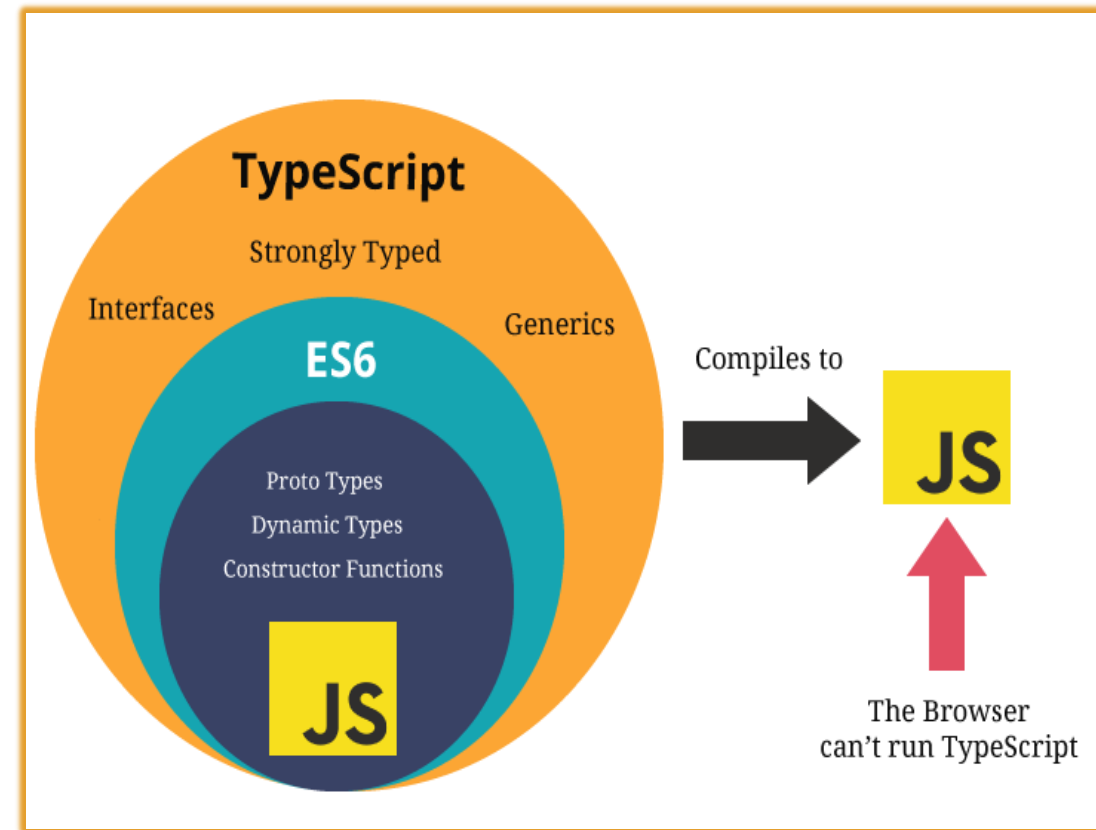
Detecting errors in code without running it is referred to as **Static Checking**. Determining what's an error (and what's not) based on the kinds of values being operated on is known as **Type Checking**.

**TypeScript** is a **Static Type Checking** language. It checks a program for errors before it's run and does so based on the types of the values.

**TypeScript** is a **Superset** of **JavaScript**. All **JavaScript** syntax is legal withing a .ts (**TypeScript**) file. (You don't need 'use strict')

Because TS and JS share syntax and runtime behavior, anything you learn about JS is helping you learn TS.

When searching for solutions to problems in TS, remember that all JS questions also apply to TS.



# TypeScript – Compiling vs Transpiling

<https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/>

<https://code.visualstudio.com/docs/typescript/typescript-compiling>

<https://www.typescriptlang.org/play>

---

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. It offers classes, modules, type checking, and interfaces to help you build robust components. TypeScript must be transpiled into JavaScript code. What's the difference between “Transpiling” and “Compiling”?

**Compiling** is the general term for taking source code written in one language and transforming into another

**Transpiling** is a specific term for taking source code written in one language and transforming into another language that has a similar level of abstraction, then compiling it into a lower level language like IL.

Both compilers and transpilers can optimize the code as part of the process. TypeScript enforces more strict typing and other rules that make JS more developer friendly.

Click [here](#) to see TS and JS compared

# TypeScript – Types

<https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html#types>

---

**TypeScript** is a typed superset of JS. This means that it adds rules about how different kinds of values can be used. TS infers the types of a values and enforces these types throughout the program.

For example, JS allows division by an empty set while TS will not. The below example in JS will print **NaN**, but TS will give an error.

```
console.log(4 / []);
```

TypeScript's type system rules are designed to allow correct programs through while still catching as many common errors as possible.

If you move some code from a JavaScript file to a TypeScript file, you might see type errors depending on how the code is written. These may be legitimate problems with the code, or TypeScript being overly conservative.



# TypeScript – Erased Types

<https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html#erased-types>

---

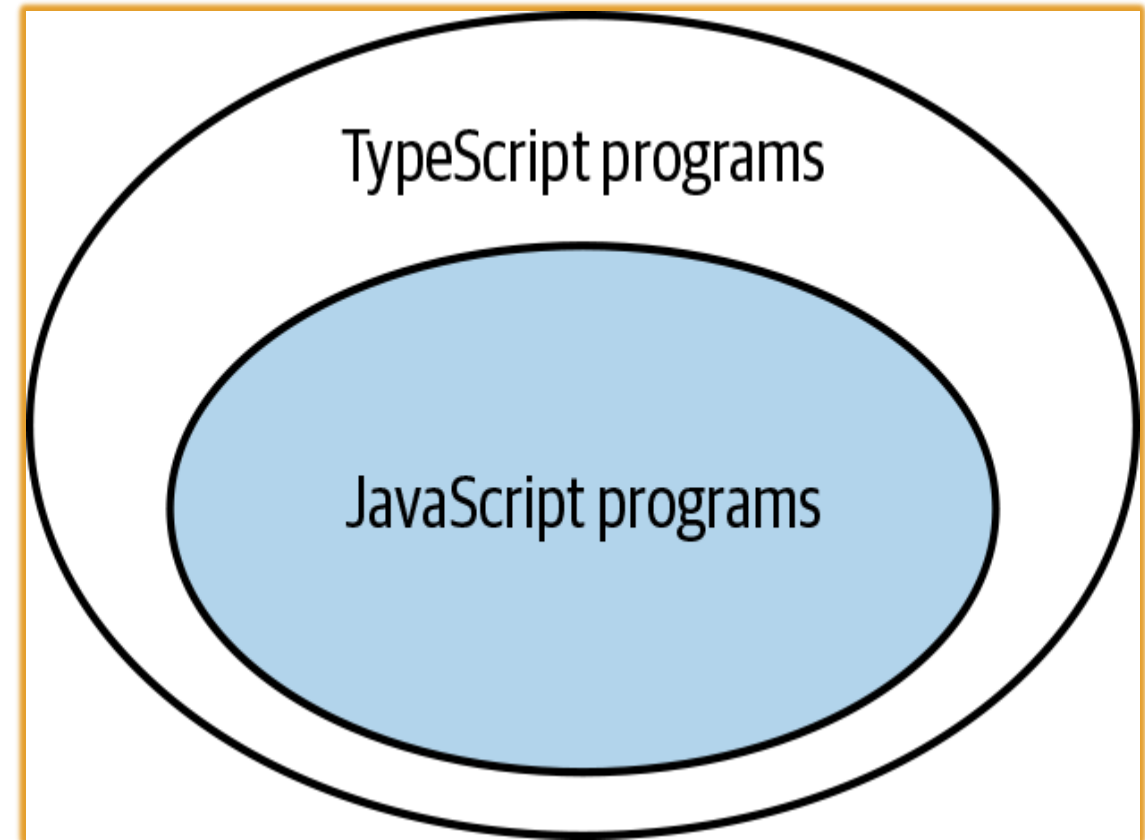
*TypeScript* (TS) preserves the runtime behavior of *JavaScript* (JS).

For example, ``100/0`` in *JS* produces *Infinity* instead of throwing a runtime exception. *TS* never changes the runtime behavior of *JS* code.

*TS*'s type system is erased. Once code is compiled, there is no persisted *type* information in the resulting *JS* code.

*TS* never changes the behavior of your program based on the *types* it inferred, so the *type* system has no bearing on how a program works once it's running.

*TS* uses *JS*'s libraries so there's no additional *TS-specific* framework to learn.



# Type Definitions

<https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html#defining-types>

---

*TS* infers most *types*. You can enforce *strict typing* by using an *interface* to declare a class and *TS* will enforce that *typing* in your *.ts* files.

Because *JS* supports classes and OOP, TypeScript does also. An *interface* declaration can also be used with classes.

There are two syntaxes for building types: *Interfaces* and *Types*. Usually, you will use *interface*. Use *type* when you need specific features.

```
interface User {  
  name: string;  
  id: number;  
}
```

```
const user: User = {  
  username: "Hayes",
```

Type '{ username: string; id: number; }' is not assignable to type 'User'.

Object literal may only specify known properties, and 'username' does not exist in type 'User'.

```
  id: 0,  
};
```

# TypeScript – Primitive Types

<https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html#defining-types>

<https://www.typescriptlang.org/docs/handbook/basic-types.html>

---

While using all JS's data types, TS extends JS's types with a few of its own.

Type	Purpose
<a href="#"><u>any</u></a>	Allow any type
<a href="#"><u>unknown</u></a>	Ensure someone using the type declares what the type is. Unknown is the type-safe counterpart of any.
<a href="#"><u>never</u></a>	Represents the type of values that never occur. EX. <i>never</i> is the return type for a function expression that always throws an exception or one that never returns.
<a href="#"><u>void</u></a>	a function which returns undefined or has no return value



# TypeScript – Type Assertions

<https://www.typescriptlang.org/docs/handbook/basic-types.html#type-assertions>

---

A “***type*** assertion” performs no special checking or restructuring of data. It has no runtime impact and is used purely by the compiler.

Type assertions have two forms. One is the “angle-bracket” syntax:

```
let someValue: any = "this is a string";  
  
let strLength: number = (<string>someValue).length;
```

And the other is the `as`-syntax:

```
let someValue: any = "this is a string";  
  
let strLength: number = (someValue as string).length;
```

# TypeScript – Structural Type System

<https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html#structural-type-system>

A core principle of *TypeScript* is that *type* checking focuses on the shape (structure) that objects have. This is called “*Structural Typing*” (or sometimes “*Duck Typing*”). The compiler only checks that at least the variable names required are present in args passed and that they match the *types* required.

```
interface Point {
```

```
  x: number;  
  y: number;  
}
```

1. Declare an interface object.

```
function printPoint(p: Point) {  
  console.log(` ${p.x}, ${p.y} `);  
}
```

2. Define a function that takes that object.

```
// prints "12, 26"
```

```
const point = { x: 12, y: 26 };  
printPoint(point);
```

3. Instantiate the object.  
Invoke the function.

```
const point3 = { x: 12, y: 26, z: 89 };  
printPoint(point3); // prints "12, 26"
```

Prints 2/3

```
const rect = { x: 33, y: 3, width: 30, height: 80 };  
printPoint(rect); // prints "33, 3"
```

Prints 2/4

```
const color = { hex: "#187ABF" };
```

```
printPoint(color);
```

ERROR!

Argument of type '{ hex: string; }' is not assignable to parameter of type 'Point'.

Type '{ hex: string; }' is missing the following properties from type 'Point': x, y

# TypeScript Interfaces

<https://www.typescriptlang.org/docs/handbook/interfaces.html>

---

Here, **LabeledValue** represents having a single property called `label` that is of *type string*. It is not required to explicitly state that the object passed into **printLabel** implements an interface (as in other languages).

In **TS**, only the objects' shape matters. If the object passed into the function meets the requirements listed, it is allowed.

**Type** checker also does not require that properties come in any specific order.

The only requirement is that property names required by the interface must be present\* AND have the required type.

```
interface LabeledValue {  
  label: string;  
}  
  
function printLabel(labeledObj: LabeledValue) {  
  console.log(labeledObj.label);  
}  
  
let myObj = { size: 10, label: "Size 10 Object" };  
printLabel(myObj);
```

\*Mark a property optional with '?' at the end of the property name.

# TypeScript – Composing Types

<https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html#composing-types>

*TypeScript* understands how code can change what type a variable could be. You can use checks to verify the type at runtime and take appropriate action.

Type	Predicate
string	<code>typeof s === "string"</code>
number	<code>typeof n === "number"</code>
boolean	<code>typeof b === "boolean"</code>
undefined	<code>typeof undefined === "undefined"</code>
function	<code>typeof f === "function"</code>
array	<code>Array.isArray(a)</code>

A Union allows you to declare what the type could be.

```
function wrapInArray(obj: string | string[]) {  
  if (typeof obj === "string") {  
    //      ^ = (parameter) obj: string  
    return [obj];  
  } else {  
    return obj;  
  }  
}
```

# TypeScript Interfaces and Class Types

<https://www.typescriptlang.org/docs/handbook/interfaces.html#class-types>

---

Interfaces explicitly enforce that a class meets a particular contract for properties and functions. In *TS*, Interfaces only describe the public side of the class.

```
interface ClockInterface {  
    currentTime: Date;  
    setTime(d: Date): void;  
}  
  
class Clock implements ClockInterface {  
    currentTime: Date = new Date();  
    setTime(d: Date) {  
        this.currentTime = d;  
    }  
    constructor(h: number, m: number) {}  
}
```

# TypeScript – Extending Interfaces

<https://www.typescriptlang.org/docs/handbook/interfaces.html#extending-interfaces>

---

Classes and interfaces can extend interfaces. This allows you to copy the members of one interface into another interface (or class). This gives you more flexibility in how you separate your interfaces into reusable components.

```
interface Shape {  
    color: string;  
}  
  
interface PenStroke {  
    penWidth: number;  
}  
  
interface Square extends Shape, PenStroke {  
    sideLength: number;  
}  
  
let square = {} as Square;  
square.color = "blue";  
square.sideLength = 10;  
square.penWidth = 5.0;
```



# TypeScript Functions

<https://www.typescriptlang.org/docs/handbook/functions.html>

In TypeScript, there are classes, namespaces, and modules, but functions still play the key role in describing how to do things. TypeScript adds some new capabilities to standard JavaScript functions.

Just as in JavaScript, TypeScript functions can be created both as a named function or as an anonymous function. They can also refer to variables outside of the function body.

You can/should explicitly **type** the parameters of the function (**IMPORTANT!!!**).

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.

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) {
    return x + y;
};
```

```
let z = 100;

function addToZ(x, y) {
    return x + y + z;
}
```

```
let myAdd: (x: number, y: number) => number = function(
    x: number,
    y: number
): number {
    return x + y;
};
```

# TypeScript Function Param Types

<https://www.typescriptlang.org/docs/handbook/functions.html#optional-and-default-parameters>

---

Contrary to JS, in TS, every function parameter is assumed to be **required** by the function.

Make a parameter **optional** by placing a '?' behind the parameter name. **Optional** parameters must be the last parameters in the list.

Give parameters **default** values with '= "value"'.

When the **default** parameter comes last, it is treated as **optional**.

**Rest** Parameters in TS are like args parameters in JS. **Rest** parameters are treated as a boundless number of **optional** parameters. The compiler builds an array of the arguments passed in with the name given after the ellipsis (...). The ellipsis is also used in the type of the function with rest parameters.

```
function buildName(firstName: string, lastName?: string) {  
    if (lastName) return firstName + " " + lastName;  
    else return firstName;  
}
```

Optional parameters

```
function buildName(firstName: string, lastName = "Smith")  
    return firstName + " " + lastName;  
}
```

Default parameters

```
function buildName(firstName: string, ...restOfName: string[]) {  
    return firstName + " " + restOfName.join(" ");  
}
```

Rest parameters

# TypeScript Classes and Inheritance

<https://www.typescriptlang.org/docs/handbook/classes.html>

Starting with ECMAScript 2015 (ECMAScript 6), JavaScript programmers can build their applications using an object-oriented, class-based approach. **TypeScript** developers can use OOP techniques, and *transpile* them into JavaScript. As in JS, TS **Abstract** classes may only be inherited.

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world");
```

```
class Animal {
  move(distanceInMeters: number = 0) {
    console.log(`Animal moved ${distanceInMeters}m.`);
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof! Woof!");
  }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

```
abstract class Animal {
  abstract makeSound(): void;
  move(): void {
    console.log("roaming the earth...");
  }
}
```

# TypeScript Type Annotations

<https://www.tutorialsteacher.com/typescript/type-annotation>

---

One of the main benefits of *TypeScript* over *JavaScript* is that you are allowed to explicitly specify the *type* of a variable. This is done with *Type Annotations*.

The *Type Annotation* is placed after the name of the variable (or parameter, property, etc)

*TypeScript* includes all the primitive types of *JavaScript*- number, string and Boolean plus adds some new ones.

```
var age: number = 32; // number variable
var name: string = "John"; // string variable
var isUpdated: boolean = true; // Boolean variable
```

```
function display(id:number, name:string)
{
    console.log("Id = " + id + ", Name = " + name);
}
```

```
var employee : {
    id: number;
    name: string;
};
```

```
employee = {
    id: 100,
    name : "John"
}
```

# TypeScript Inheritance with *this*

<https://www.typescriptlang.org/docs/handbook/classes.html#inheritance>

Each *derived* class that contains a constructor function must call *super()* to execute the constructor of the *base* class.

Before a property on *this* is accessed from within a constructor body, *super()* must be called.

This is an important rule that TypeScript will enforce.

```
class Animal {
  name: string;
  constructor(theName: string) {
    this.name = theName;
  }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) {
    super(name);
  }
  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters);
  }
}

class Horse extends Animal {
  constructor(name: string) {
    super(name);
  }
  move(distanceInMeters = 45) {
    console.log("Galloping...");
    super.move(distanceInMeters);
  }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

# TypeScript – Class Property Modifiers

<https://www.typescriptlang.org/docs/handbook/classes.html#public-private-and-protected-modifiers>

In TypeScript, each class member is **public** by default.

TypeScript supports the new JavaScript syntax for **private** fields and has its own way to declare a member as being marked **private**. Private fields cannot be accessed from outside of their containing classes.

Members declared **protected** can be accessed from within their class and **deriving** classes. A constructor may also be marked **protected**. This means that the class cannot be instantiated outside of its containing class but can be **extended**.

**Readonly** properties must be initialized at their declaration or in the constructor.

```
class Animal {  
  private name: string;  
  constructor(theName: string) {  
    this.name = theName;  
  }  
}
```

```
class Animal {  
  #name: string;  
  constructor(theName: string) { this.#name = theName; }  
}
```

```
class Person {  
  protected name: string;  
  protected constructor(theName: string) {  
    this.name = theName;  
  }  
}  
  
// Employee can extend Person  
class Employee extends Person {  
  private department: string;  
  
  constructor(name: string, department: string) {  
    super(name);  
    this.department = department;  
  }  
  
  public getElevatorPitch() {  
    return `Hello, my name is ${this.name} and I work in ${this.department}.`;  
  }  
}  
  
let howard = new Employee("Howard", "Sales");  
let john = new Person("John"); // Error: The 'Person' constructor is protected
```



# TypeScript – Static Class Properties

<https://www.typescriptlang.org/docs/handbook/classes.html#static-properties>

---

**Static** members of a class are visible on the class itself rather than on the instances. Each instance accesses this shared value through prepending the name of the class.

```
class Grid {  
  static origin = { x: 0, y: 0 };  
  calculateDistanceFromOrigin(point: { x: number; y: number }) {  
    let xDist = point.x - Grid.origin.x;  
    let yDist = point.y - Grid.origin.y;  
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;  
  }  
  constructor(public scale: number) {}  
}  
  
let grid1 = new Grid(1.0); // 1x scale  
let grid2 = new Grid(5.0); // 5x scale  
  
console.log(grid1.calculateDistanceFromOrigin({ x: 10, y: 10 }));  
console.log(grid2.calculateDistanceFromOrigin({ x: 10, y: 10 }));
```

# TypeScript Modules

<https://www.typescriptlang.org/docs/handbook/modules.html>

---

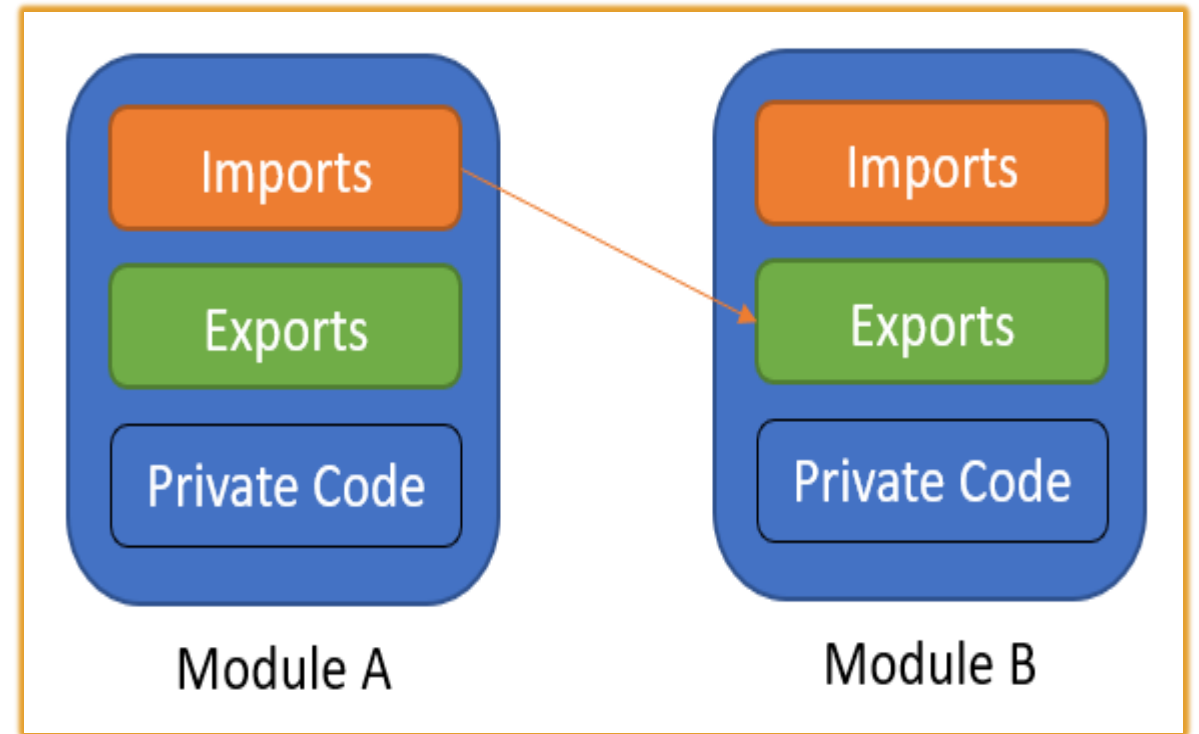
TypeScript shares the JS concept of *Modules*.

**Modules** in TS have their own scope. Anything declared inside a *module* is not visible outside that *module* unless it is explicitly *exported*.

To consume a property *exported* from a different *module*, it must be *imported* using an *import* method.

The relationships between *modules* are specified in terms of *imports* and *exports* at the file level.

In TypeScript, 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 (and therefore in *modules* as well).



# TypeScript - Exporting a Declaration

<https://www.typescriptlang.org/docs/handbook/modules.html#export>

---

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the **export** keyword.

First, use the **export** keyword to make a class, function, or variable available to other **modules** from within the module (component).

Second, **import** the class, function, or variable into the **module** (component) where you want to implement it.

```
export interface StringValidator {  
    isAcceptable(s: string): boolean;  
}
```

```
import { StringValidator } from "./StringValidator";  
  
export const numberRegex = /^[0-9]+$/;  
  
export class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegex.test(s);  
    }  
}
```

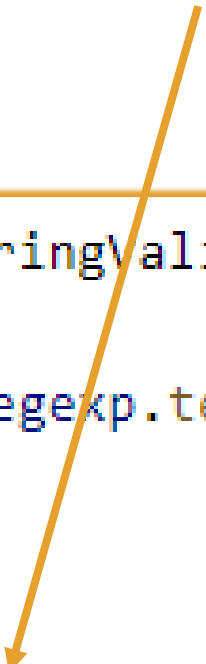
# TypeScript - Export

<https://www.typescriptlang.org/docs/handbook/modules.html#export-statements>

---

*Export* Statements allow you to rename the statement you want to export.

```
class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegex.test(s);  
    }  
}  
  
export { ZipCodeValidator };  
export { ZipCodeValidator as mainValidator };
```



# TypeScript Imports

<https://www.typescriptlang.org/docs/handbook/modules.html#import>

---

```
import { ZipCodeValidator } from "../ZipCodeValidator";  
  
let myValidator = new ZipCodeValidator();
```

Imports can also be renamed.

```
import { ZipCodeValidator as ZCV } from "../ZipCodeValidator";  
let myValidator = new ZCV();
```

# Create a TS version of GuessingGame Setup

<https://www.valentinog.com/blog/typescript/>

---

1. Create a new folder for this project in your repo2.
2. Make sure you have Node.js with `node -v` in Command Line. If not, go to [nodejs.org](https://nodejs.org) to get it.
3. In Command Line, run `npm init -y` to create a *package.json* file.
4. In Command Line run `npm i typescript --save-dev` to install a *TS* dependency via *npm* (this installs for just this program).
5. Configure the node script to compile with *tsc*. In the new *package.json* file, include `"scripts": { "tsc": "tsc" },`
6. Run `npm run tsc -- --init` in Command Line to create a *tsconfig.json* file for which the TS compiler (*tsc*) will look. You should get `"message TS6071: Successfully created a tsconfig.json file."` in the Command Line.
7. Replace all the original content of the *tsconfig.json* file with `{ "compilerOptions": { "target": "es5", "strict": true } }` **ES5** is the newest JS release. **Strict** enforces *TS*'s highest level of strictness and *tsc* will insert `"use strict"` atop each *JS* file.
8. Compile and run with `npm run tsc` in Command Line. This will transpile the TS code to JS code and create a file in the same folder.
9. Complete the [Migrating from JavaScript](#) tutorial.
10. Make sure to use `<script>` to include the new .js file.