



TypeScript Fundamentals

.NET

TypeScript is a static typechecker for JavaScript programs.

A static typechecker is a tool run on code before the code itself is run to ensure that the data types will be consistent at runtime.

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

TypeScript vs ES6

<https://www.geeksforgeeks.org/difference-between-es6-and-typescript>

ES6 (aka, ECMAScript 2015) is the latest version of JavaScript. It allows developers to instantiate an object using the **new** operator or an arrow function. Also, the **return** keyword can be omitted.

TypeScript is an open-source, object-oriented programming language developed and maintained by Microsoft. It was launched and introduced on October 1st, 2012.

TypeScript does not directly run on any browser. It requires a compiler to transpile **.ts** files into **.js** files.

ES6	TYPESCRIPT
ECMA script 6 is the sixth edition of ECMAScript trademarked scripting language specification defined by ECMA international.	TypeScript is a free and open-source pure object-oriented programming language developed and maintained by Microsoft.
It does not support all data types.	It supports all primitive data types.
It was introduced in 2015.	It was introduced in 2012.
It does not support these features.	It contains features such as generics and type annotations, Inference, Enums, and Interfaces.
In ES6 modules can be used by importing and exporting.	Modules are of 2 types: Internal and external modules
It is comparatively more flexible in development time.	It is to eradicate the development errors
It has two scopes, Global Scope, Local Scope.	It has three scopes, Global Scope, Class Scope, Local Scope.
The list of companies that use ES6 is Slack, StackShare, eBay, Asana, Intuit, Swat.io.	The list of companies that use TypeScript is Slack, Asana, CircleCI, Intuit, Swat.io, Avocode.

TypeScript –

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

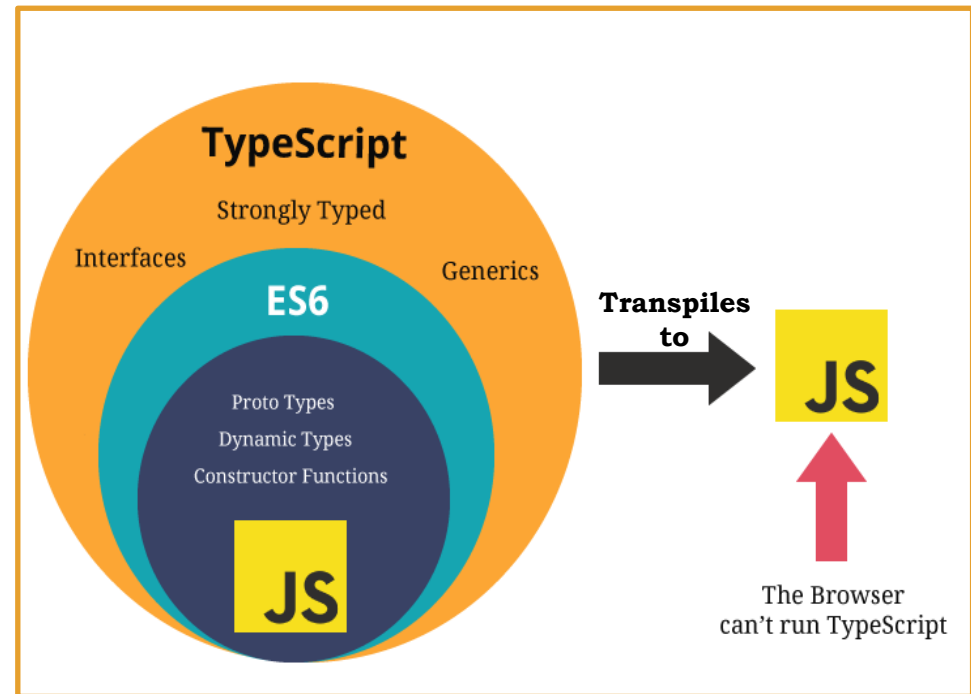
The relationship between TypeScript (TS) and JavaScript (JS) is unique among modern programming languages.

TypeScript checks JavaScript programs for errors before execution. It does so based on the *kinds (types) of values*. TypeScript is a **static type checker**.

TypeScript is a ***typed superset*** of JavaScript. This means that it ‘transpiles’ to plain JavaScript.

TypeScript enforces strict ***typing*** along with other rules. It has classes, modules, ***type*** checking, and interfaces.

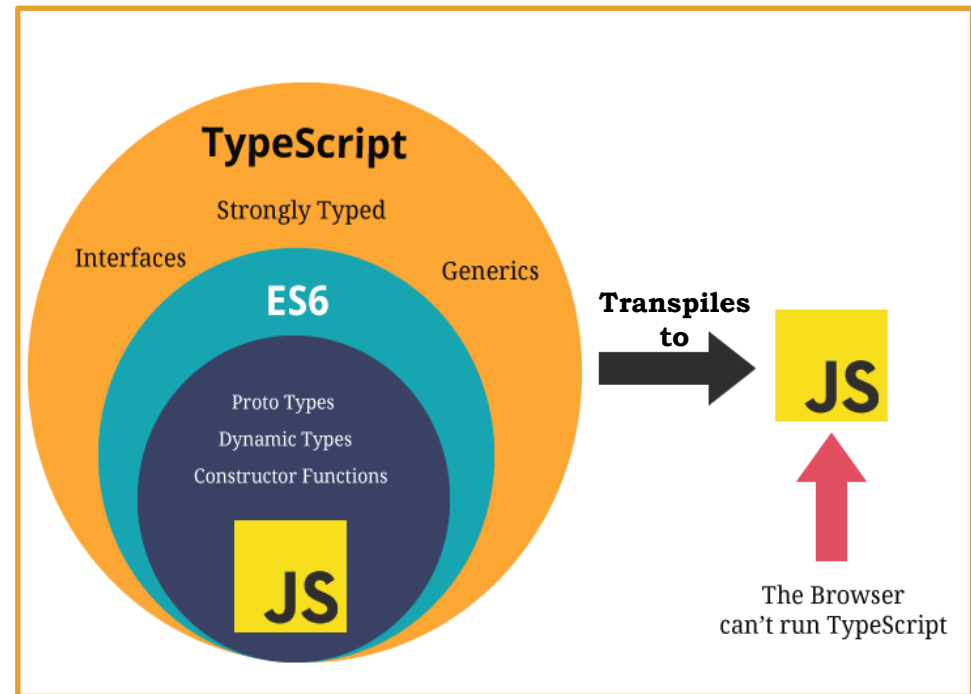
TypeScript must be ***transpiled*** into JavaScript code to be run.



TypeScript – Overview

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

- **Static Checking*** – The process of detecting errors in code without running it.
- TypeScript is a **Static Type Checking** language. It checks a program for errors before it's run based on the **types** of the values.
- TypeScript is a **Superset** of JavaScript. All JavaScript syntax is legal within a **.ts** (TypeScript) file.
- You don't need **'use strict'**.
- All JavaScript rules also apply to TypeScript.
- There are no browsers that can run TypeScript. It must be **'transpiled'** to JS.



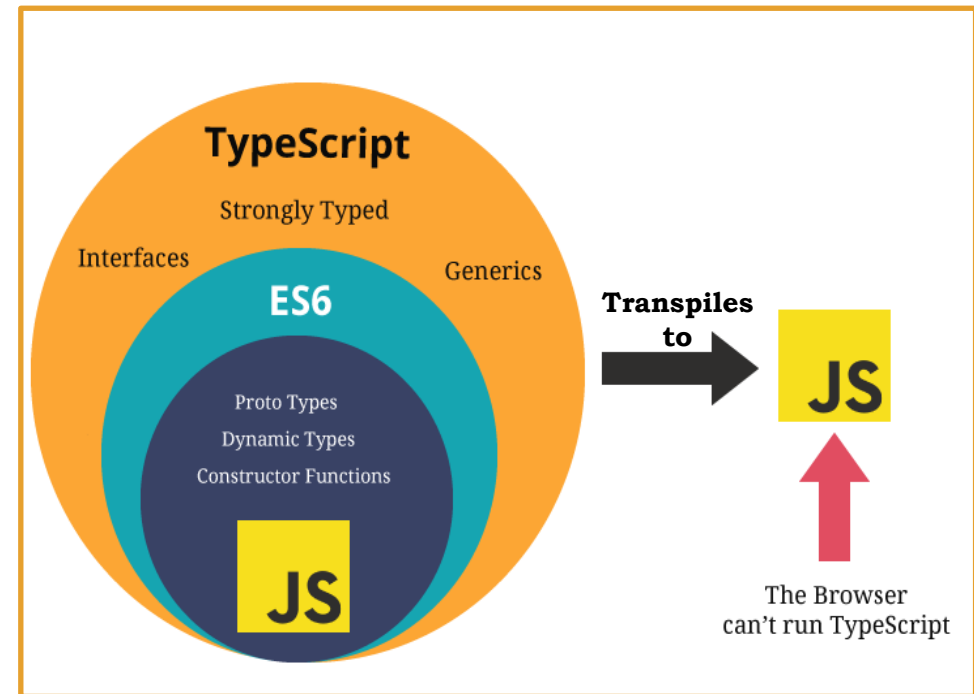
*Dynamic Type Checking is when type checking is done at runtime

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>

“Transpiling” vs. “Compiling”

- **Compiling** is the term for taking source code written in one language and transforming it into another, more basic, language.
- **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.



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

TypeScript – Types

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

TypeScript adds rules about how different value **types** can be used. TypeScript can infer value **types** and will enforce explicit and inferred **types** throughout a program.

TypeScript's **type** system imposes restrictions that are designed to allow correct programs through, while catching as many common errors as possible.

If you transpile code from a JavaScript file to a TypeScript file, **type** errors that are legitimate problems with the code are noted, but it could also be that TypeScript is being overly conservative.

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

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.

```
const user = {  
  firstName: "Angela",  
  lastName: "Davis",  
  role: "Professor"  
}
```

```
console.log(user.name)
```

```
Property 'name' does not exist on type '{ firstName: string;  
lastName: string; role: string; }'.
```

TypeScript Type Annotations

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

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

One of the primary benefits of TypeScript over JavaScript is that variable **types** can be explicitly specified. This is done with **Type Annotations** (Type Assertions).

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

TypeScript has all the primitive **types** of JavaScript 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): void
{
    console.log("Id = " + id + ", Name = " + name);
}
```

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

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

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;
```


Type Definitions

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

- **Type Definitions** (also known as **Type Assertions**) are used to define the type for a variable or object.
- TypeScript supports classes and OOP.
- There are two syntaxes for building types: **Interfaces** and **Types**.
- TypeScript infers most **types**, but you can enforce **strict typing** by using an **interface** to declare a class. TypeScript will enforce the **typing** declared in the interface.
- Conventionally, **interface** is used more often.
- Use **type** when specific properties on objects need to be enforced.

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

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

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

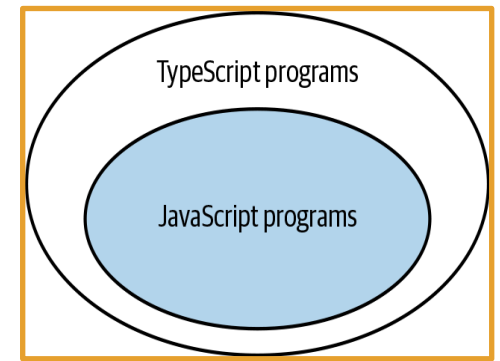
Object literal may only specify known properties, and 'username' does not exist in type 'User'.

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

The above interface is declared and later an object is instantiated based on the **User** interface. TypeScript will throw an error if a property called username is inserted.

TypeScript – Erased Types

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



- Due to Typescript's **type** annotations there are no browsers that can run TypeScript itself.
- TypeScript has its own compiler that strips out (erases) TypeScript-specific code so that it can be run as JavaScript.
- After transpilation, no **type** information in the resulting **JS** code is left.
- TypeScript preserves the runtime behavior of JavaScript.
- Typescript never changes the behavior of a program based on the **types** inferred. This means the **type** system has no influence on how a program works once it's running.
- Typescript uses JavaScript libraries so there's no additional Typescript-specific framework to learn.

```
1 // @showEmit
2 function greet(person: string, date: Date) {
3   console.log(`Hello ${person}, today is ${date.toDateString()}!`);
4 }
5
6 greet("Maddison", new Date());
```

TypeScript

JavaScript

```
"use strict";
// @showEmit
function greet(person, date) {
  console.log(`Hello ${person}, today is ${date.toDateString()}!`);
}
greet("Maddison", new Date());
```

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>

TypeScript uses all JavaScripts data *types*. TypeScript extends JavaScript *types* with a few of its own.

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

Duck-Typing (Structural Type System)

<https://www.javatpoint.com/typescript-duck-typing>

‘Duck-Typing’ is a method/rule used to check the type compatibility for more complex types.

TypeScript uses ‘duck-typing’ to compare objects. It checks that both objects have the same matching names and types.

If two objects have different properties, functions, or **types**, the TypeScript compiler will generate a compile-time error.

- If it walks like a duck, flies like a duck, and quacks like a duck...
- Feature selection:
 - Walking
 - Flying
 - Quacking



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**” (“**Duck Typing**”). The compiler only checks that (at minimum) the variable names required are present in arguments 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.

4. Invoke the function.

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

Prints 2 of the
3 parameters

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

Prints 2 of the
4 parameters

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

ERROR Result

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 – Composing Types

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

Because JavaScript has *loose typing*, the *type* of a variable in TypeScript code may need to be verified at runtime so the code can take appropriate action.

Type	Predicate*
string	<code>typeof myString === "string"</code>
number	<code>typeof myNum === "number"</code>
boolean	<code>typeof myBool === "boolean"</code>
undefined	<code>typeof undef === "undefined"</code>
function	<code>typeof myFunc === "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;  
  }  
}
```

*Often used in a if-else comparison.

TypeScript Interfaces and Class Types

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

Interfaces are a great way to explicitly enforce that a class meets a particular contract for properties and functions.

In TypeScript, Interfaces only describe the public properties and fields of a 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 Classes and Inheritance

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

TypeScript developers can use OOP techniques like inheritance.

Abstract classes in TypeScript and JavaScript 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 Inheritance with *this*

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

As in JavaScript, each *derived* class that contains a constructor function must call **super()** to execute the constructor of the *base(parent, super...)* class and before any property on **this** is accessed from within the constructor body.

This is a 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 members can be **private**.
- TypeScript supports the new JavaScript syntax for **private** fields.
- **Private** fields cannot be accessed from outside of their containing classes.
- **Protected members** can be accessed from within their class and **deriving** classes.
- A **protected class constructor** 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 class 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 only accessible on the class itself. Not on the instances.

Each class instance accesses this shared value through prepending the name of the containing 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 Interfaces

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

- Here, **LabeledValue** is an interface with a string property, **label**.
- It is not required to explicitly state that the object passed into a function implements an interface (as in C#).
- In **TS**, only the objects' **shape** matters. If the argument passed into the function meets the requirements listed (the **shape**), it is allowed.
- **Type** checking 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 – Extending Interfaces

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

Classes and Interfaces can extend other Interfaces.

This allows you to copy the members of one interface into another interface or class.

```
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>

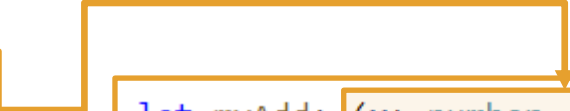
In *TypeScript*, there are classes, namespaces, and modules. *TypeScript* adds new capabilities to JS, but *functions* still play the key role in describing how to complete actions.

TypeScript functions can be *named* or *anonymous*. They can also refer to variables outside of the function body, but in scope.

You must explicitly **type** the parameters of functions.

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 **types** are required.

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



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

TS Function Parameter Types

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

- 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 last.
- Give parameters **default** values with 'paramName = "value"'.
• When a **default** parameter comes last, it is treated as **optional**.
- **Rest** Parameters ('...paramName') in **TS** are like **args** parameters in **JS**.
- **Rest** parameters are treated as **optional** parameters. The compiler builds an array of the additional arguments passed with the name given after the ellipsis (...).

```
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 Modules

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

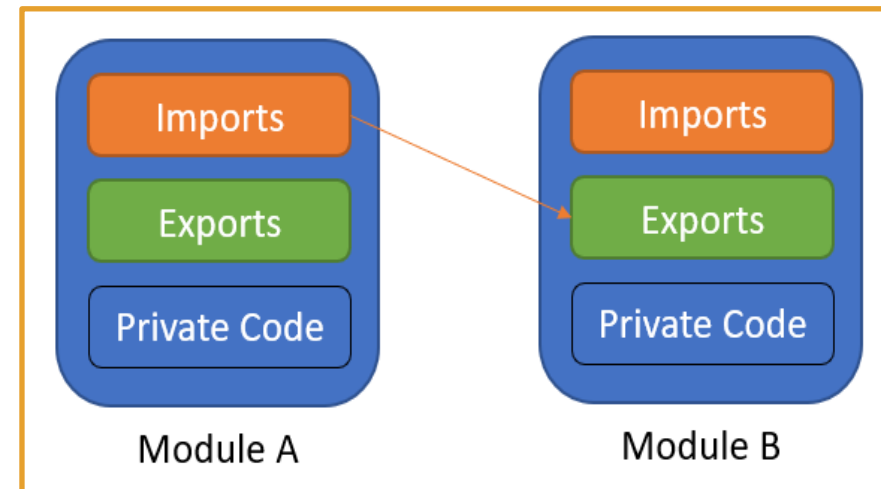
TypeScript shares the JavaScript concept of **Modules**. **Modules** in TypeScript have their own scope. A module must be explicitly **exported** to make its members visible.

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**.

As in JavaScript, a TypeScript file without any top-level **import** or **export** declarations is treated as a script whose contents are available in the global scope and in **modules** as well.



TypeScript - Exporting a Declaration

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

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

1. Use **export** to make a class, function, or variable available to other *modules*.
2. Use an **import** statement in a *module (component)* to gain access to a class, function, or variable what has been exported.

```
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 and **import** statements allow you to rename a *module*.

Conventionally, **import** statements are listed at the top of the document while **export** statements are listed at the bottom.

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

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

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


Create a TS version of GuessingGame (1/2)

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

<https://www.typescriptlang.org/docs/handbook/asp-net-core.html>

1. Create a new folder for this project in your repo.
2. Make sure you have Node.js with `node -v` in Command Line. If not, go to 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` (dash-dash) to install a TypeScript dev dependency via `npm` (`--save-dev` installs for just this program, in dev mode). (why `--save-dev`?)
5. In the new `package.json` file, add to the node script commands a command to compile with `tsc`. Add `"scripts":{ "tsc": "tsc"}`. `"scripts"` should already be among the key:value pairs.
6. Run `npm run tsc -- --init` (dash-dash, space, dash-dash) 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.

Create a TS version of GuessingGame (2/2)

<https://www.valentinog.com/blog/typescript/>
<https://www.typescriptlang.org/docs/handbook/asp-net-core.html>

7. Replace all the original content of the `tsconfig.json` file with:
 - `{ "compilerOptions": { "target": "es5", "module": "es2015", "strict": true } }`
 - [Why “es2015”?](#)
8. ES6 is the newest JS release. “strict” enforces the highest level of TypeScript strictness. Visit <https://aka.ms/tsconfig.json> for info on the `tsconfig.json` file.
9. Create a `.ts` file in the same folder and add `.ts` code to it. Or complete the [Migrating from JavaScript](#) tutorial.
10. Compile and run with `npm run tsc` in Command Line. This will transpile the TypeScript code to JavaScript code and create a like-named `.js` file in the same folder.
11. Make sure to use `<script type=“module” src=“jsFileName.js”>` to include the new `.js` file inside your `.html`.