TypeScript Fundamentals

.NET

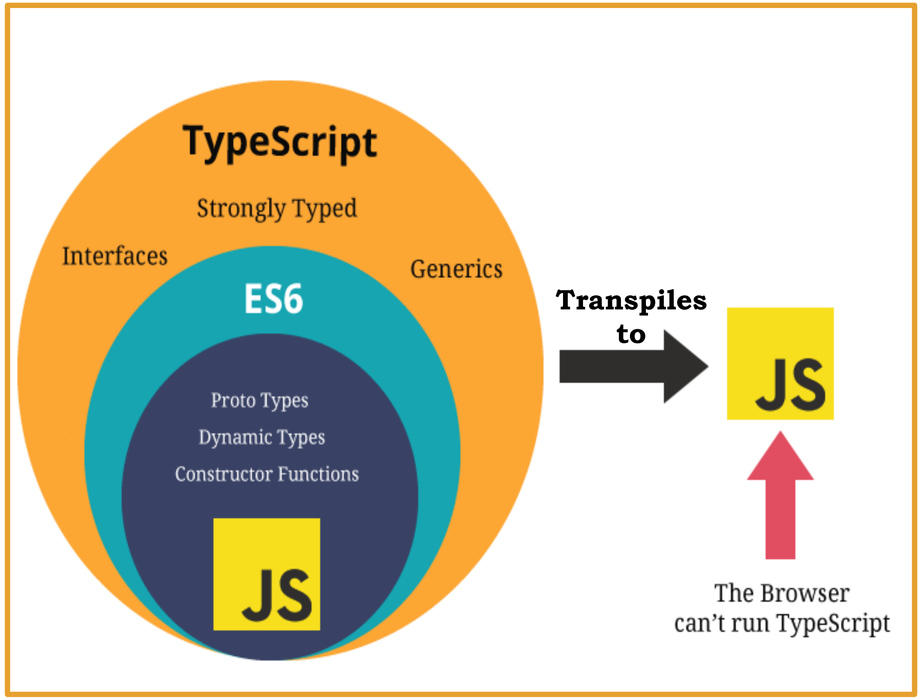
TypeScript is a superset of JavaScript and a static typechecker of JavaScript programs. A static typechecker is a tool run on code before the code itself is run to ensure that the data types will be consistent at runtime.

HTTPS://WWW.TYPESCRIPTLANG.ORG/DOCS/HANDBOOK/INTRO.HTML#ABOUT-THIS-HANDBOOK

TypeScript – Overview

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

- ***Static Checking*** - Detecting errors in code without running it.
- ***Type Checking*** - Determining what code is an error (and what's not) based on the data types being operated on.
- ***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***.

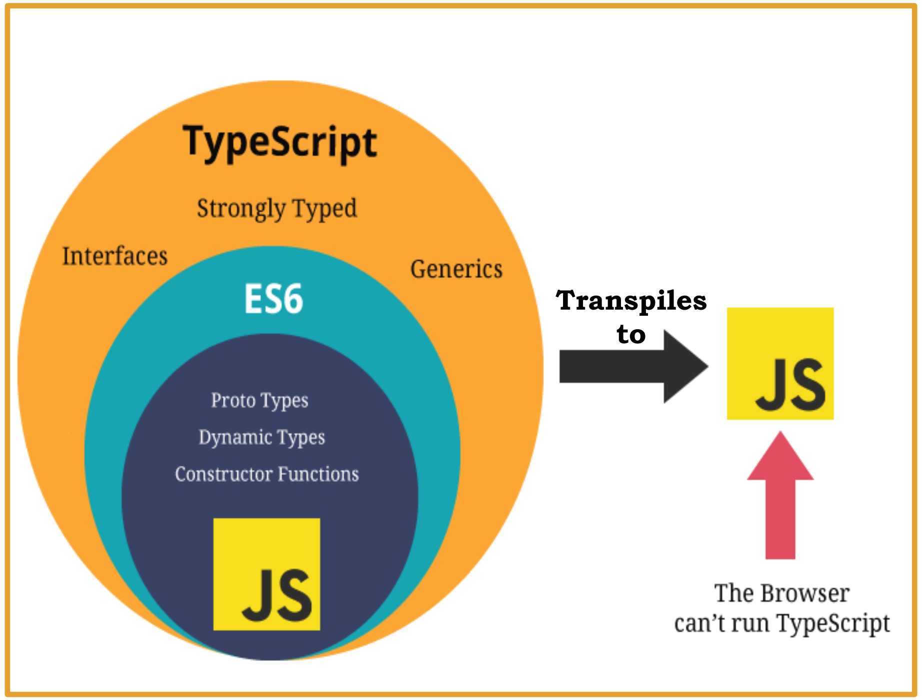


*Dynamic Type Checking is when type checking is done at runtime

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

TypeScript is a typed superset of ***JavaScript***. This means that it 'transpiles' to plain ***JavaScript***. ***TypeScript*** enforces strict typing, among other rules. It has classes, modules, ***type*** checking, and interfaces. ***TypeScript*** must be ***transpiled*** into ***JavaScript*** code to be run.



"Transpiling" vs. "Compiling"

- ***Compiling*** is the 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.

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. ***TS*** also can infer value ***types*** and will enforce those ***types*** throughout a program. ***TypeScript's type*** system restrictions are designed to allow correct programs through, while catching as many common errors as possible.

```
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.

If you move code from a ***JavaScript*** file to a ***TypeScript*** file, you might see type errors that are legitimate problems with the code or it may be that ***TypeScript*** is being overly conservative.

```
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; }'.
```

Duck-Typing

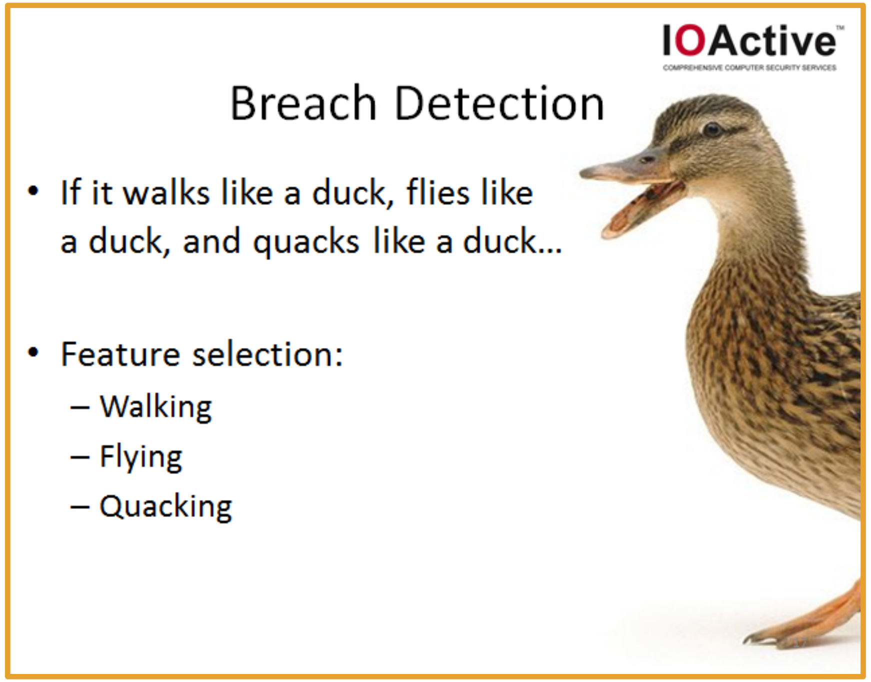
https://www.javatpoint.com/typescript-duck-typing

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

"If it walks like a duck... If it talks like a duck..."

TypeScript uses 'duck-typing' to compare one object with other 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.



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)
{
    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

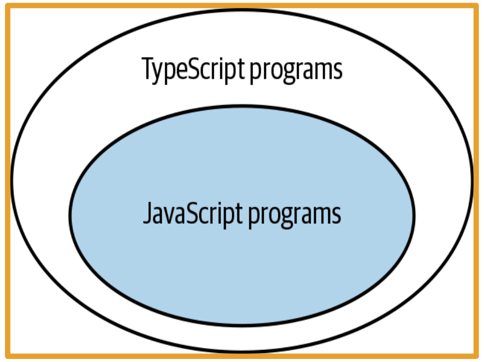
- **TS** supports classes and OOP.
- There are two syntaxes for building types: ***Interfaces*** and ***Types***.
- ***TS*** infers most ***types***, but you can enforce ***strict typing*** by using an ***interface*** to declare a class. ***TS*** will enforce the ***typing*** declared in the interface.
- Conventionally, ***interface*** is used more often. 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 – 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 in order to strip out (erase) ***TypeScript***-specific code so that it can be run as JavaScript.
- There is no persisted ***type*** information in the resulting *JS* code.
- ***TypeScript*** preserves the runtime behavior of ***JavaScript***.
- *TS* never changes the behavior of your program based on the ***types*** it inferred, so the ***type*** system has no influence on how a program works once it's running.
- *TS* uses *JS* libraries so there's no additional *TS-specific* framework to learn.

TypeScript

```
// @showEmit
function greet(person: string, date: Date) {
  console.log(`Hello ${person}, today is ${date.toDateString()}!`);
}

greet("Maddison", new Date());
```

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 *JS's* data types. *TS* extends *JS* types with a few of its own.

Type	Purpose
any	Allow any type
unknown	Ensure someone using the type declares what the type is. Unknown is the type-safe counterpart of any.
never	Represents the type of values that never occur. EX. ***never*** is the return type 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

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 "***Duck Typing***"). The compiler only checks that at least 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 JS has loose typing, you may need to use type checks to verify the ***type*** of a variable in you TS code at runtime and take appropriate action.

Type	Predicate*
string	typeof myString === “string”
number	typeof myNum === “number”
boolean	typeof myBool === “boolean”
undefined	typeof undef === “undefined”
function	typeof myFunc === “function”
array	Array.isArray(a)

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 *TS*, 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. As in ***JavaScript***, ***Abstract*** classes in ***TypeScript*** 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*** class.

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

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.
- *TS* has away to declare a ***private*** member.
- *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 visible on the class itself rather than 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, and ***TypeScript*** adds some new capabilities to JS, but ***functions*** still play the key role in describing how to complete actions.

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

You should 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 parts 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

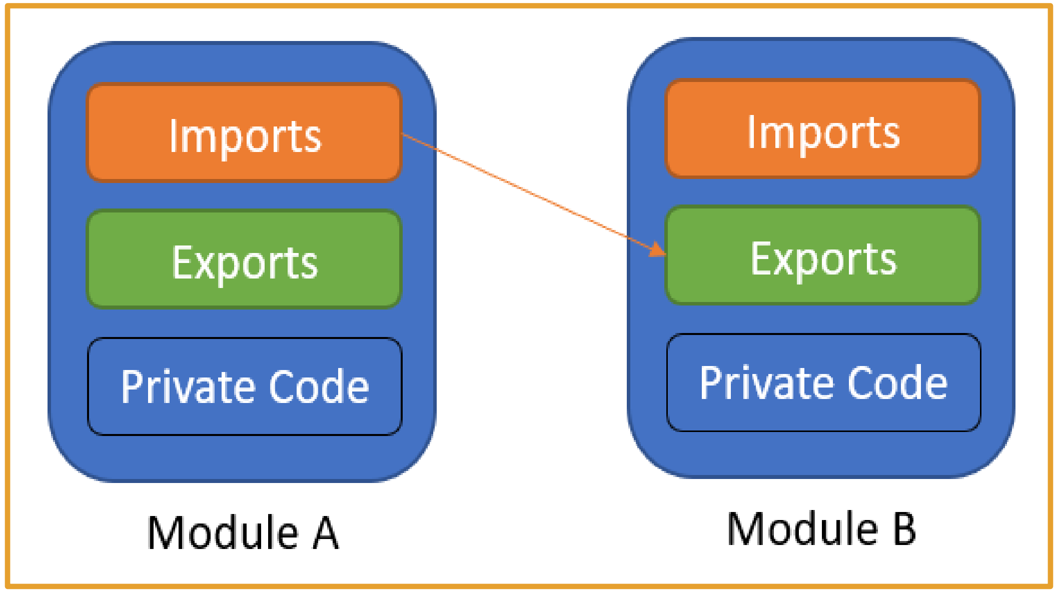
TS shares the *JS* concept of ***Modules***. ***Modules*** in *TS* have their own scope. A module must be explicitly exported to make its members visible.

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 *TS*, any file containing a top-level ***import*** or ***export*** is considered a ***module***.

As in JS, a TS 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, 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 numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.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

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) (what is –save-dev?) to install a ***TS*** dependency via ***npm*** (this installs for just this program).
5. In the new ***package.json*** file, change the node script to compile with ***tsc***. Include “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.
7. Replace all the original content of the ***tsconfig.json*** file with: { "compilerOptions": { "target": "es5", "strict": true } }
8. *ES5* is the newest JS release. “strict” enforces ***TS’s*** highest level of strictness. Visit https://aka.ms/tsconfig.json for info on the tsconfig file
9. 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.
10. Complete the Migrating from JavaScript tutorial.
11. Make sure to use <script src=“jsFileName.js”> to include the new .js file inside your .html.