



# **3416-003 Web Development Using Angular 7 and TypeScript**

## **Module 2: Typescript**



## Module 2: Section 1

# What Is This Course About



# Learning Outcomes for this Module

- What is TypeScript
- Type Safety
- TypeScript The Compiler
- TypeChecker
- TypeScript Vs JavaScript
- TypeScript Variable
- How Are Types Bound
- Are Types Automatically Converted
- When Are Types Checked
- When Are Errors Surfaced
- All About Types

# What is Typescript?

- TypeScript is the language that will power the next generation of web apps, mobile apps, NodeJS projects, and Internet of things(IoT) devices. It will make your programs safer by checking for common mistakes, serve as documentation for yourself and future engineers and make refactoring painless. Typescript will double your productivity as a programmer.
- What exactly do I mean when I say “safer”. What I am talking about, of course, is type safety.

# TYPE SAFETY

- Using types to prevent programs from doing invalid things.
- Here are a few examples of things that are invalid:
  - Multiplying a number and a list
  - Calling a function with a list of strings when it actually needs a list of objects
  - Calling a method on an object when that method doesn't actually exist on that object
  - Importing a module that was recently moved.
- Run example1.ts by typing in dos-prompt or terminal(for mac): `tsc example1.ts`

# TYPE SAFETY

- Example1.ts:

```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd\

C:\>cd C:\workspace-UT-Angular8-JAN-2020\Session2\Examples

C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example1.ts
example1.ts:1:1 - error TS2365: Operator '+' cannot be applied to types '3' and 'undefined[]'.

1 3 + []; //Error: TS2365: Operator '+' cannot be applied to types '3' and 'never[]'.
   ~~~~~

example1.ts:4:5 - error TS2339: Property 'foo' does not exist on type '{}'.

4 obj.foo;    // Error TS2339: Property 'foo' does not exist on type '{}'
   ~~~~

example1.ts:10:3 - error TS2345: Argument of type '"z"' is not assignable to parameter of type 'number'.

10 a("z");    // Error TS2345: Argument of type '"z"' is not assignable to
   ~~~~

Found 3 errors.

C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>
```

# TypeScript – The Compiler

- Depending on what programming languages you worked in the past, you'll have a different understanding of how programs work. The way TypeScript works is unusual compared to other mainstream languages like JavaScript or Java.
- Programs are files that contain a bunch of text written by you, the programmer. That text is parsed by a special program called a compiler, which transforms it into an abstract syntax tree(AST), a data structure that ignores things like whitespace, comments, and where you stand on the tabs versus spaces debate. The compiler then converts that abstract syntax tree(AST) to a lower-level representation called bytecode. You can feed that bytecode into another program called a runtime to evaluate it and get a result. So when you run a program, what you're really doing is telling the runtime to evaluate the bytecode generated by the compiler from the AST parsed from your source code. The details vary, but for most languages this is an accurate high-level view.

# TypeScript – The Compiler

- Once again, the steps are:
  - Program is parsed into an abstract syntax tree(AST)
  - abstract syntax tree(AST) is compiled to bytecode.
  - Bytecode is evaluated by the runtime.
- Where TypeScript is special is that instead of compiling straight to bytecode. **TypeScript compiles to ..JavaScript code! You then run the JavaScript code like you normally would – in your browser.**
- Previously we mention that TypeScript makes my code safer! When does that happen?
- Great question. I actually skipped over a crucial step: after the TypeScript Compiler generates an AST for your program – but before it emits code – it typechecks your code.



# TypeScript – TYPECHECKER

- A special program that verifies that your code is typesafe.
- This typechecking is the magic behind TypeScript. It's how TypeScript makes sure that your program works as you expect, that there aren't obvious mistakes.
- So if we include typechecking and JavaScript emission, the process of compiling TypeScript now looks roughly like this:

	1. TypeScript source -> TypeScript AST
TS	2. AST is checked by typechecker
	3. TypeScript AST -> JavaScript source
	4. JavaScript source --> JavaScript AST
JS	5. AST -> bytecode
	6. Bytecode is evaluated by runtime

# TypeScript – TYPECHECKER

- A set of rules that a typechecker uses to assign types to your program.
- There are generally two kinds of type systems: type systems in which everything you have to tell the compiler what type everything is with explicit syntax, and type systems that infer the types of things for you automatically. Both approaches have trade-offs.
- TypeScript is inspired by both kinds of type systems: you can explicitly annotate your types, or you can let TypeScript infer most of them for you.

# TypeScript – TYPECHECKER

- To explicitly signal to TypeScript what your types are, use annotations. Annotations take the form `value: type` and tell the typechecker, “Hey! You see this value here? Its type is `type`.” Let’s look at a few examples (the comments following each line are the actual types inferred by TypeScript):
- Compiled and run Example2.tsc:

Command to compile `typescript:tsc example2.ts`

Command to run `typescript:node example2.js`

# TypeScript Versus JavaScript

- Let's take a deeper look at TypeScript's type system, and how it compares to JavaScript's type system:

Type system feature			JavaScript	TypeScript	
Are types automatically converted			Yes	No	
When are types checked?			At runtime	At compile time	
When are errors surfaced?			At runtime (mostly)	At compile time(mostly)	

# TypeScript - Variable

- TypeScript follows the same rules as JavaScript for variable declarations. Variables can be declared using: var, let, and const.
- Var Declaration
  - Variables in TypeScript can be declared using var keyword, same as in JavaScript. The scoping rules remains the same as in JavaScript.
- Let Declaration
  - To solve problems with var declarations, ES6 introduced two new types of variable declarations in TypeScript, using the keywords let and const. TypeScript, being a superset of JavaScript, also supports these new types of variable declarations.
  - Example: Variable Declaration using let

```
1 let employeeName = "John";  
2 // or  
3 let employeeName:string = "John";
```

# TypeScript - Variable

- The let declarations follow the same syntax as var declarations. Unlike variables declared with var, variables declared with let have a block-scope. This means that the scope of let variables is limited to their containing block, e.g. function, if else block or loop block. Consider example2a.ts.

# TypeScript - Variable

- example2a.ts.

```
1  let num1:number = 1;
2
3  function letDeclaration() {
4      let num2:number = 2;
5
6      if (num2 > num1) {
7          let num3: number = 3;
8          num3++;
9      }
10
11     while(num1 < num2) {
12         let num4: number = 4;
13         num1++;
14     }
15
16     console.log(num1); //ok
17     console.log(num2); //ok
18     console.log(num3); //Compiler Error: Cannot find name 'num3'
19     console.log(num4); //Compiler Error: Cannot find name 'num4'
20
21 }
22
23 letDeclaration();
```

# TypeScript - Variable

- Command: `tsc example2a.ts`.

```
1 Ram

C:\AlbertLam\Angular7-UT-May2019\Session5\Examples>tsc example2a.ts
example2a.ts:18:17 - error TS2304: Cannot find name 'num3'.

18     console.log(num3); //Compiler Error: Cannot find name 'num3'
                        ~~~~~

example2a.ts:19:17 - error TS2304: Cannot find name 'num4'.

19     console.log(num4); //Compiler Error: Cannot find name 'num4'
                        ~~~~~

Found 2 errors.
```



# TypeScript - Variable

- Advantages of using let over var.
- 1.)Block-scoped let variables cannot be read or written to before they are declared. See example2b.ts.
- Run tsc example2b.ts, node example2b.js

```
TS example2b.ts ▸ ...
1 console.log(num1); //Compiler Error: error TS2448: Block-scoped variable 'num' used before its declaration
2 let num1:number = 10;
3
4 console.log(num2); //ok, outut: undefined
5 var num2:number = 10;
```

```
C:\AlbertLam\Angular7-UT-May2019\Session5\Examples>tsc example2b.ts
example2b.ts:1:13 - error TS2448: Block-scoped variable 'num1' used before its declaration.

1 console.log(num1); //Compiler Error: error TS2448: Block-scoped variable 'num' used before its declaration
    ~~~~~
    'num1' is declared here.

example2b.ts:2:5
2 let num1:number = 10;
    ~~~~~
    'num1' is declared here.

Found 1 error.
```

# TypeScript - Variable

- Example2b.js:

```
1 console.log(num1); //Compiler Error: error TS2448: Block-scoped variable 'num' used before its declaration
2 var num1 = 10;
3 console.log(num2); //ok, output: undefined
4 var num2 = 10;
5
```

- node example2b.js

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example2b.js
undefined
undefined
```

# TypeScript - Variable

- In example2b.ts, the TypeScript compiler will give an error if we use variables before declaring them using let, whereas it won't give an error when using variables before declaring them using var.

# TypeScript - Variable

- Let variables cannot be re-declared.
  - The TypeScript compiler will give an error when variables with the same name(case sensitive) are declared multiple times in the same block using let. See example2c.tsc
  - Command:tsc example2c.ts

```
TS example2c.ts ▸ ...
1  let num:number = 1; //OK
2  let Num:number = 2; //OK
3  let NUM:number = 3; //OK
4  let NuM:number = 4; //OK
5
6  let num:number = 5; //Compiler Error: Cannot redeclared block-scoped variable 'num'
7  let Num:number = 6; //Compiler Error: Cannot redeclared block-scoped variable 'Num'
8  let NUM:number = 7; //Compiler Error: Cannot redeclared block-scoped variable 'NUM'
9  let NuM:NUMBER = 8; //Compiler Error: Cannot redeclared block-scoped variable 'NuM'
```

# TypeScript - Variable

- Let variables cannot be re-declared.
  - example2c.tsc

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example2c.ts
example2c.ts:1:5 - error TS2451: Cannot redeclare block-scoped variable 'num'.

1 let num:number = 1; //OK
   ~~~~

example2c.ts:2:5 - error TS2451: Cannot redeclare block-scoped variable 'Num'.

2 let Num:number = 2; //OK
   ~~~~

example2c.ts:3:5 - error TS2451: Cannot redeclare block-scoped variable 'NUM'.

3 let NUM:number = 3; //OK
   ~~~~

example2c.ts:4:5 - error TS2451: Cannot redeclare block-scoped variable 'NuM'.

4 let NuM:number = 4; //OK
   ~~~~

example2c.ts:6:5 - error TS2451: Cannot redeclare block-scoped variable 'num'.

6 let num:number = 5; //Compiler Error: Cannot redeclared block-scoped variable 'num'
   ~~~~

example2c.ts:7:5 - error TS2451: Cannot redeclare block-scoped variable 'Num'.

7 let Num:number = 6; //Compiler Error: Cannot redeclared block-scoped variable 'Num'
   ~~~~

example2c.ts:8:5 - error TS2451: Cannot redeclare block-scoped variable 'NUM'.

8 let NUM:number = 7; //Compiler Error: Cannot redeclared block-scoped variable 'NUM'
   ~~~~
```

# TypeScript - Variable

- Let variables cannot be re-declared.
  - In example2c.tsc, the TypeScript compiler treats variable names as case sensitive. “num” is different than “Num”, so it won’t give any error. However, it will give an error for the variables with the same name and case.
  - Variables with the same name and case can be declared in different block as shown in example2d.tsc

```
TS example2d.ts ▸ ...
1  let num:number = 1;
2
3  function demo() {
4      let num:number = 2;
5
6      if(true) {
7          let num:number = 3;
8          console.log(num); //Output: 3
9      }
10     console.log(num); //Output: 2
11 }
12
13 console.log(num); //Output: 1
14 demo();
```

# TypeScript - Variable

- Let variables cannot be re-declared.
  - example2d.tsc

```
C:\AlbertLam\Angular7-UT-May2019\Session5\Examples>tsc example2d.ts
C:\AlbertLam\Angular7-UT-May2019\Session5\Examples>node example2d.js
1
3
2
```

# TypeScript - Variable

- Similarly, the compiler will give an error if we declare a variable that was already passed in as an argument to the function as shown in example2e.ts
- Thus, variables declared using let minimize the possibilities of runtime errors, as the compiler give compile-time errors. This increases the code readability and maintainability.

```
TS example2e.ts ▶ letDemo
1  function letDemo(a: number) {
2      let a:number = 10; //Compiler Error: TS2300: Duplicate identifier 'a'
3      let b:number = 20;
4
5      return a + b;
6  }
```



# TypeScript - Variable

- Example2e.ts:

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example2e.ts
example2e.ts:1:18 - error TS2300: Duplicate identifier 'a'.

1 function letDemo(a: number) {
                  ~

example2e.ts:2:9 - error TS2300: Duplicate identifier 'a'.

2   let a:number = 10; //Compiler Error: TS2300: Duplicate identifier 'a'
    ~

Found 2 errors.
```

# TypeScript Versus JavaScript(HOW ARE TYPES BOUND)

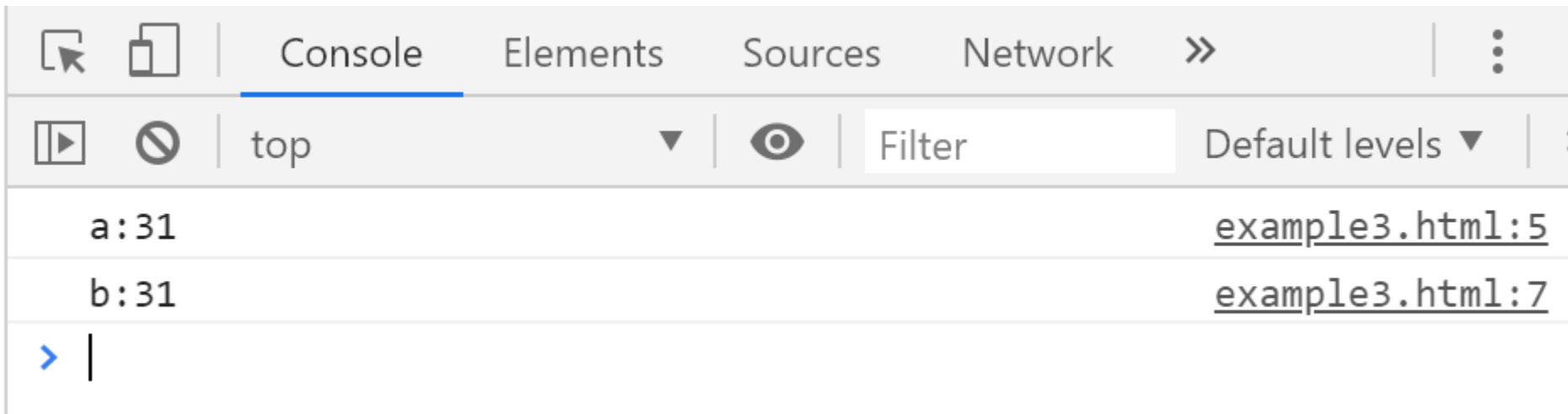
- HOW ARE TYPES BOUND?
  - Dynamic type binding means that JavaScript needs to actually run your program to know the types of things in it. JavaScript doesn't know your types before running your program.
  - TypeScript is a gradually typed language. That means that TypeScript works best when it knows the types of everything in your program at compile time.

# TypeScript Versus JavaScript(ARE TYPES AUTOMATICALLY CONVERTED)

- ARE TYPES AUTOMATICALLY CONVERTED?
  - JavaScript is weakly typed, meaning if you do something invalid like add a number and an array, it will apply a bunch of rules to figure out what you really meant so it can do the best it can with what you gave it. Let's walk through the specific example of how JavaScript evaluates `3 + [1]`:
    1. JavaScript notices that 3 is a number and `[1]` is an array.
    2. Because we're using `+`, it assumes we want to concatenate the two.
    3. It implicitly converts 3 to a string, yield "3".
    4. It implicitly converts `[1]` to a string, yielding "1".
    5. It concatenates the results, yielding "31".

# TypeScript Versus JavaScript(ARE TYPES AUTOMATICALLY CONVERTED)

- We could do this more explicitly too (so JavaScript avoids doing steps 1,3, and 4).
- Run example3.html



# TypeScript Versus JavaScript(ARE TYPES AUTOMATICALLY CONVERTED)

- While JavaScript tries to be helpful by doing clever conversions for you. TypeScript complains as soon as you do something invalid. When you run that same JavaScript through TSC, you'll get an error.
- Run `example4.ts:tsc example4.ts`

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example4.ts
example4.ts:1:9 - error TS2365: Operator '+' cannot be applied to types '3' and 'number[]'.

1  var a = 3 + [1];                                //evalutate to "31"
    ~~~~~
Found 1 error.
```

# TypeScript Versus JavaScript(ARE TYPES AUTOMATICALLY CONVERTED)

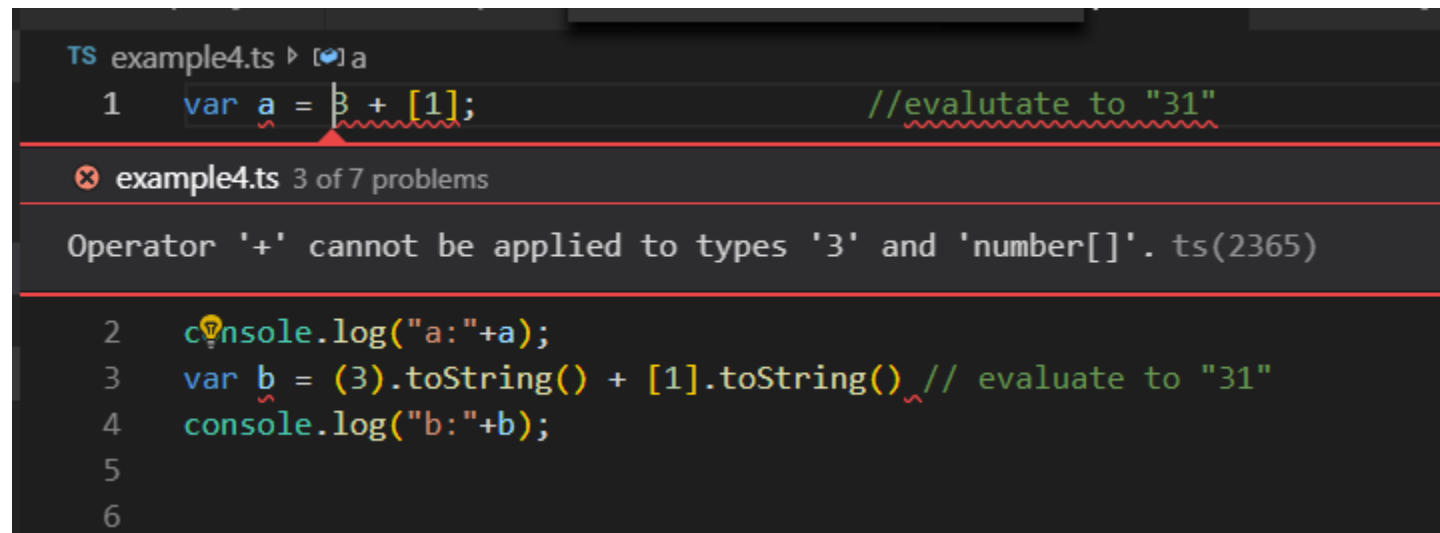
- If you do something that doesn't seem right, TypeScript complains, and if you're explicit about your intentions. TypeScript gets out of your way. This behavior makes sense: who in their right mind would try to add a number and an array, expecting the result to be a string.
- The kind of implicit conversion that JavaScript does can be a really hard-to-track-down source of errors. It makes it hard for individual engineers to get their jobs done, and it makes it even harder to scale code across a large team, since every engineer needs to understand the implicit assumptions your code makes.
- In short, if you must convert types, do it explicitly.

# TypeScript Versus JavaScript(WHEN ARE TYPES CHECKED)

- WHEN ARE TYPES CHECKED?
  - In most places JavaScript doesn't care what types you give it, and it instead tries to do its best to convert what you gave it to what it expects.
  - TypeScript, on the other hand, typechecks your code at compile time so you don't need to actually run your code to see the ERROR from the previous example. TypeScript statically analyzes your code for errors like these, and shows them to you before you run it. If your code doesn't compile, that's a really good sign that you made a mistake and you should fix it before you try to run the code.

# TypeScript Versus JavaScript(WHEN ARE TYPES CHECKED)

- WHEN ARE TYPES CHECKED?
  - Figure shows what happens when I type the last code example into VSCODE:



The screenshot shows a VS Code editor window with a file named 'example4.ts'. The code is as follows:

```
1  var a = 3 + [1]; //evaluate to "31"
2  console.log("a:"+a);
3  var b = (3).toString() + [1].toString(); // evaluate to "31"
4  console.log("b:"+b);
5
6
```

A red squiggly line under the '+' operator in line 1 indicates an error. Below the code, a red bar shows the error message: 'Operator '+' cannot be applied to types '3' and 'number[]'. ts(2365)'. The error message is followed by the file name 'example4.ts' and the count '3 of 7 problems'.



# TypeScript Versus JavaScript(WHEN ARE TYPES CHECKED)

- WHEN ARE TYPES CHECKED?
  - With a good TypeScript extension for your preferred code editor, the error will show up as a red squiggly line under your code as you type it. This dramatically speeds up the feedback loop between writing code, realizing that you made a mistake, and updating the code to fix that mistake.

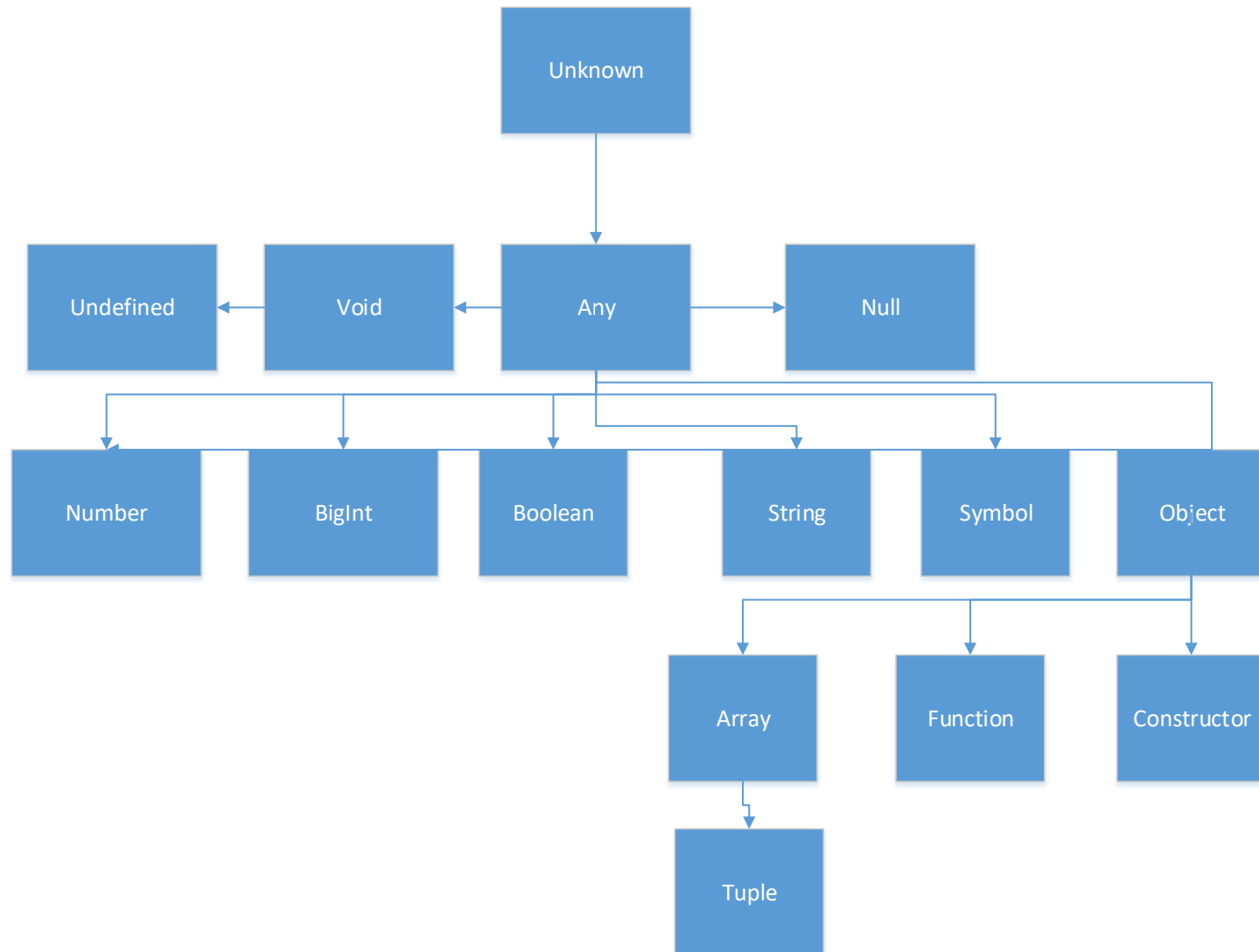
# TypeScript Versus JavaScript(WHEN ARE ERRORS SURFACED)

- WHEN ARE ERRORS SURFACED?
  - When JavaScript throws exceptions or performs implicit type conversions, it does so at runtime? This means you have to actually run your program to get a useful signal back that you did something invalid. In the best case, that means as part of a unit test; in the worst case, it means an angry email from a user.
  - TypeScript throws both syntax-related errors and type-related errors at compile time. In practice, that means those kinds of errors will show up in your code editor, right as you type – it's an amazing experience if you've never worked with an incrementally compiled statically typed language before.
  - That said, there are lots of errors that TypeScript can't catch for you at compile time –things like broken network connections, and malformed user inputs – that will result in runtime exceptions. What TypeScript does to make compile-time errors out of most errors that would have otherwise been runtime errors in a pure JavaScript world.

# All About Types

- TYPE is a set of values and the things you can do with them.
- If that sounds confusing, let me give a few familiar examples:
  - The Boolean type is the set of all Booleans (there are just two: true and false) and the operations you can perform on them (like ||, &&, and !).
  - The number type is the set of all numbers and the operations you can perform on them (like +, -, \*, /, %, ||, &&, and ?), including the methods you can call on them like .toFixed, .toPrecision, .toString, and so on.
  - The string type is the set of all strings and the operations you can perform on them (like +, ||, and &&), including the methods you can call on them like .concat and .toUpperCase.
- Remember, the whole point is to use the typechecker to stop you from doing invalid things. And the way the typechecker knows what's valid and what's not is by looking at the types you're using and how you're using them.

# All About Types



# Talking About Types

- When programmers talk about types, they share a precise, common vocabulary to describe what they mean. We're going to use this vocabulary throughout this course.
- Say you have a function that takes some values and returns that value multiplied by itself. Run example5.html.
- Clearly, this function will only work for numbers – if you pass anything besides a number to squareOf, the result will be invalid. So what we do is explicitly annotate the parameter's type.

```
1 <html>
2 <head>
3 <script>
4   function squareOf(n) {
5     return n * n
6   }
7
8   console.log(squareOf(1)); // evaluates to 1
9   console.log(squareOf('z')); //evaluates to NAN
10 </script>
11 </head>
12 <body>
13
14 </body>
15 </html>
```

# Talking About Types

- Compile example6.ts.
- Command:tsc example6.ts

```
1 function squareOf(n: number): number {  
2     return n * n;  
3 }  
4  
5 console.log(squareOf(2)); // evalutes to 2  
6 console.log(squareOf("z")); //evaluates to NAN  
7
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example6.ts  
example6.ts:6:22 - error TS2345: Argument of type '"z"' is not assignable to parameter of type 'number'.  
  
6 console.log(squareOf("z")); //evaluates to NAN  
                        ~~~~~  
  
Found 1 error.
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example6.js  
4  
NaN
```

# Talking About Types

- Now if we call `squareOf` with anything but a number, TypeScript will complain right away. This is a trivial example, but it's enough to introduce a couple of concepts that are key to talking about types in TypeScript. We can say the following things about the last code example:
  1. `squareOf`'s parameter `n` is constructed to number.
  2. `squareOf`'s return type is number.
  3. The type of the value `2` is assignable to (equivalently: compatible with) number.

# Talking About Types

- Without a type annotation, `squareOf` is unconstrained in its parameter, and you can pass any type of argument to it. Once we constrain it, TypeScript goes to work for us verifying that every place we call our function, we call it with a compatible argument. In this example the type of `2` is `number`, which is assignable to `squareOf`'s annotation `number`, so TypeScript accepts our code; but `'z'` is a `string`, which is not assignable to `number`, so TypeScript complains.
- You can also think of it in terms of bounds: we told TypeScript that `n`'s upper bound is `number`, so any value we pass to `squareOf` has to be at most a `number`. If it's anything more than a `number` (like, if it's a value that might be a `number` or might be a `string`), then it's not assignable to `n`.



# The ABCs of Types

- Let's take a tour of the types TypeScript supports, what values they contain, and what you can do with them. We'll also cover a few basic language features for working with types: type aliases, union types, and intersection types.
- `Type:any`
- “any” is the GodFather of types. It does anything for a price, but you don't want to ask any for a favor unless you're completely out of options. In TypeScript everything needs to have a type at compile time, and any is the default type when you (the programmer) and TypeScript(the typechecker) can't figure out what type something is. It's a last resort type, and you should avoid it when possible.

# The ABCs of Types

- Why should you avoid it? Remember what a type is? (It's a set of values and the things you can do with them). Any is the set of all values, and you can do anything with any. That means that if you have a value of type any you can add to it, and multiply by it.

# The ABCs of Types

- “any” make your value behave like it would in regular JavaScript, and totally prevents the typechecker from working its magic. When you allow any into your code you’re flying blind. Avoid any like fire, and use it only as a very, very last resort.
- Compile:tsc example7.ts
- Run:node example7.js

```
- [C:\workspace-UT-Angular8-JAN-2020\Session2\Examples\example7.ts]
Edit View Search Document Project Tools Window Help
Clipboard
ce-UT-Angular8
2
les
de
1 let a: any = 666; //any
2 console.log("a:"+a);
3 let b: any = ['danger']; //any
4 console.log("b:"+b);
5 let c = a + b; //any
6 console.log("c:"+c);
7
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example7.ts
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example7.js
a:666
b:danger
c:666danger
```

# The ABCs of Types

- Notice how the third type, in `example7.ts`, should report an error (why are you trying to add a number and an array?), but doesn't because you told TypeScript that you're adding two anys. If you want to use `any`, you have to be explicit about it. When TypeScript infers that some value is of type `any` (for example, if you forgot to annotate a function's parameter, or if you imported an untyped JavaScript module), it will throw a compile-time exception and toss a red squiggly at you in your editor. By explicitly annotating `a` and `b` with the `any` type (`: any`), you avoid the exception – it's our way of telling TypeScript that you know what you're doing.

# Boolean

- The Boolean type has two values: true and false. You can compare them (with ==, ===, ||, &&, and ?), negate them (with !), and not much else. Use Boolean like this in Example8.ts:

```
ts example8.ts ...
1  let a = true;           //boolean
2  var b = false;          //boolean
3  const c = true;         //true
4  let d: boolean = true;  //boolean
5  !t e: true = true;      // true
6  let f: true = false;    // Error TS2322: Type 'false' is not assignable to type 'true'
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example8.ts
example8.ts:6:5 - error TS2322: Type 'false' is not assignable to type 'true'.
6  let f: true = false; // Error TS2322: Type 'false' is not assignable to type 'true'
   ~

Found 1 error.
```

# Boolean

- Example8.ts shows a few ways to tell TypeScript that something is Boolean:
  1. You can let TypeScript infer that your value is a Boolean (a and b).
  2. You can let TypeScript infer that your value is a specific Boolean (c).
  3. You can tell TypeScript explicitly that your value is a Boolean (d)
  4. You can tell TypeScript explicitly that your value is a specific Boolean (e and f).

# number

- Number is the set of all numbers: integers, floats, positives, negatives, Infinity, NaN, and so on. Numbers can do, well, numbery things, like addition (+), subtraction (-), modulo(%),

rtLam\Angular7-UT-Sept2019\Session2\Examples\example9.ts]

```
Search Document Project Tools Window Help
-----+-----1-----+-----2-----+-----3-----+-----4-----+-----5-----+-----6-----+-----
1 let a = 1234; // number
2 var b = Infinity * 0.10; // number
3 const c = 5678; // 5678
4 let d = a < b; //boolean
5 let e: number = 1000; // number
6 let f: 26.218 = 26.218; //26.218
7 let g: 26.218 = 10; // Error TS2322: Type '10' is not assignable
8 // to type '26.218'.
9
10
11 console.log("a:"+a);
12 console.log("b:"+b);
13 console.log("c:"+c);
14 console.log("d:"+d);
15 console.log("e:"+e);
16 console.log("f:"+f);
17 console.log("g:"+g);
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example9.ts
example9.ts:7:5 - error TS2322: Type '10' is not assignable to type '26.218'.
```

```
7 let g: 26.218 = 10; // Error TS2322: Type '10' is not assignable
```

Found 1 error.

# number

- Number is the set of all numbers: integers, floats, positives, negatives, Infinity, NaN, and so on. Numbers can do, well, numbery things, like addition (+), subtraction (-), modulo(%), and comparison(<). Let's look at example9.ts

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example9.ts
example9.ts:7:5 - error TS2322: Type '10' is not assignable to type '26.218'.
```

```
7 let g: 26.218 = 10; // Error TS2322: Type '10' is not assignable
   ~
```

```
Found 1 error.
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example9.js
```

```
a:1234
```

```
b:Infinity
```

```
c:5678
```

```
d:true
```

```
e:1000
```

```
f:26.218
```

```
g:10
```



# number

- Like in the Boolean example, there are four ways to type something as a number:
  1. You can let TypeScript infer that your value is a number (a and b and d).
  2. You can use const so TypeScript infers that your value is a specific number( c ).
  3. You can tell TypeScript explicitly that your value is a number( e).
  4. You can tell TypeScript explicitly that your value is specific number (f and g).

# bigint

- BigInt is a newcomer to JavaScript and TypeScript: it lets you work with large integers without running into rounding errors. While the number type can only represent whole numbers up to 2 to the power of 53, bigint can represent integers bigger than that too. The bigint type and supports things like addition (+), subtraction (-), multiplication (\*), division (/), and comparison (<). See example10.ts

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example10.ts
example10.ts:5:9 - error TS1353: A bigint literal must be an integer.

5 let e = 88.5n; // Error TS1353: A bigint literal must be an integer.
    ~~~~~

Found 1 error.
```

# string

- String is the set of all strings and the things you can do with them like concatenate (+), slice (.slice), and so on. See example11.ts

```

; - [C:\AlbertLam\Angular7-UT-Sept2019\Session2\Examples\example11.ts]
Edit View Search Document Project Tools Window Help
ClipText
s
am
r7-UT-Sept2019
n2
ples
ode
processor.js
processor.ts
processor2.js
processor2.ts
1 let a = 'hello'; //string
2 let b = 'billy' //string
3 const c = '!'; // '!'
4 let d = a + ' ' + b + c //string
5 let e: string = 'zoom'; //string
6 console.log("a:" + a);
7 console.log("b:" + b);
8 console.log("c:" + c);
9 console.log("d:" + d);
10 console.log("e:" + e);
11
12
```

# string

- String is the set of all strings and the things you can do with them like concatenate (+), slice (.slice), and so on. See example11.ts

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example11.js  
a:hello  
b:billy  
c:!  
d:hello billy!  
e:zoom
```

# Functions(Declaring and Invoking Functions).

- In JavaScript, functions are first-class objects. That means you can use them exactly like you would any other object: assign them to variables, pass them to other functions, return them from functions, assign them to objects and prototypes, write properties to them, read those properties back, and so on. There is a lot you can do with functions in JavaScript and TypeScript models all of those things with its rich type system.

# Functions(Declaring and Invoking Functions).

- See example12.ts:

```
TS example12.ts ▸ ...
1  function add1(a, b) {
2      |   return a + b;
3  }
4
5  function add2(a: number, b: number) {
6      |   return a + b;
7  }
8
9  function add3(a: number, b: number): number {
10     |   return a + b;
11 }
12
13 console.log(add1(1,2));
14 console.log(add2(2,2));
15 console.log(add3(3,3));
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example12.js
```

```
3
4
6
```

# Functions(Declaring and Invoking Functions).

- Here's what a function looks like in TypeScript  
example12.ts.
- You will usually explicitly annotate function parameters(a and b in this example) – TypeScript will always infer types throughout the body of your function. The return type is inferred, but you can explicitly annotate it too if you want.

# Functions(Declaring and Invoking Functions).

- Example12.ts used named function syntax to declare the function, but JavaScript and TypeScript support at least four ways to do this. See example13.ts.

rtLam\Angular7-UT-Sept2019\Session2\Examples\example13.ts]

Search Document Project Tools Window Help

```
1 // Named function
2 function greet1(name: string): string {
3     return "hello " + name;
4 }
5
6 // function expression
7 let greet2 = function(name: string): string {
8     return "hello " + name;
9 }
10
11 //Arrow function expression
12 let greet3 = (name: string): string => {
13     return "hello " + name;
14 }
15
16 //Shorthand arrow function expression
17 let greet4 = (name: string): string =>
18     "hello " + name;
19
20 console.log(greet1("albert"));
21 console.log(greet2("frances"));
22 console.log(greet3("jaden"));
23 console.log(greet4("lorraine"));
24
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node
hello albert
hello frances
hello jaden
hello lorraine
```



# Functions(Declaring and Invoking Functions).

- A quick refresher on terminology:
  - A parameter is a piece of data that a function needs to run, declared as part of a function declaration. Also called a formal parameter.
  - An argument is a piece of data you passed to a function when invoking it. Also called an actual parameter.

# Functions(Declaring and Invoking Functions).

- When you invoke a function in TypeScript, you don't need to provide any additional type information – just pass in some arguments, and TypeScript will go to work checking that your arguments are compatible with the types of your function's parameters. Type: tsc example14.ts

```
TS example14.ts ▸ ...
1  function add(a: number, b: number): number {
2      |   return a + b;
3      |
4      |
5  function greet(name: string) {
6      |   return "hello " + name;
7      |
8  add(1); //Error S2554: Expected 2 arguments, but got 1
9  greet(1, 'a'); // Error TS2345: Argument of type '*a*' is not assignable to parameter of type 'number'
10
```

# Functions(Declaring and Invoking Functions).

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example14.ts
example14.ts:8:1 - error TS2554: Expected 2 arguments, but got 1.
8 add(1); //Error S2554: Expected 2 arguments, but got 1
  ~~~~~

example14.ts:1:25
1 function add(a: number, b: number): number {
  ~~~~~
  An argument for 'b' was not provided.

example14.ts:9:9 - error TS2554: Expected 1 arguments, but got 2.
9 greet(1,'a'); // Error TS2345: Argument of type '*a*' is not assignable to parameter of type 'number'
  ~~~~~

Found 2 errors.
```

# Optional and Default Parameters

- You can use ? To mark parameters as optional. When declaring your function's parameters, required parameters have to come first, followed by optional parameters.  
Command: tsc example15.ts

```
TS example15.ts ▸ ...
1  function log(message: string, userId?: string) {
2      let time = new Date().toLocaleTimeString();
3      console.log(time, message, userId || 'Not signed in');
4  }
5
6  log('Page loaded'); //Logs "12 PM Page loaded Not signed in"
7  log('User signed in', 'da763be'); // logs User signed in da763b3
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example15.ts
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example15.js
```

```
5:07:02 PM Page loaded Not signed in
```

```
5:07:02 PM User signed in da763be
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>
```

# Arrow functions

- 'fat arrow' – syntax: `=>`
- If function has one argument, do not use parentheses
- For multiple arguments, use parentheses
- Curly braces are required for multiple lines of code, otherwise not required and return is implicit.
- Returning an object requires parentheses around return statement.

# Arrow functions

- In example16.ts, sum2 is an arrow function. (x:number, y:number) denotes the parameter types, : number specifies the return type. The fat arrow => separates the function parameters and the function body. The right side of => can contain one or more code statements.

tLam\Angular7-UT-Sept2019\Session2\Examples\example16.ts]

Search Document Project Tools Window Help

```
1 let sum1 = function (x: number, y: number): number {
2     return x + y;
3 }
4
5 let sum2 = (x: number, y: number): number => {
6     return x + y;
7 }
8
9 console.log(sum1(10,10));
10 console.log(sum2(20,20));
```

# Arrow functions

- In example16.ts will be converted to example16.js when using command: `tsc example16.ts`

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example16.ts  
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example16.js  
20  
40
```

# Arrow functions(without parameters)

- example17.ts is an arrow function without parameters. Furthermore, if the function body consists of only one statement then no need for the curly brackets and the return keyword.

bertLam\Angular7-UT-Sept2019\Session2\Examples\example17.ts]

ew Search Document Project Tools Window Help

```
1 let Print1 = function () {
2     console.log("Print1, Hello TypeScript");
3 }
4 |
5 let Print2 = () => console.log("Print2, Hello TypeScript");
6
7 let sum = (x: number, y: number) => x + y;
8
9 Print1();
10 Print2();
11 console.log(sum(3,4));
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example17.ts
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example17.js
```

```
Print1, Hello TypeScript
```

```
Print2, Hello TypeScript
```



# Arrow functions(Arrow Function in Class)

- A class can include an arrow function as a property, see example18.ts

```
1  class Employee {  
2      empCode: number;  
3      empName: string;  
4  
5      constructor(code: number, name: string) {  
6          this.empName = name;  
7          this.empCode = code;  
8      }  
9      display = () => console.log(this.empCode + " " + this.empName);  
10  
11 }  
12 let emp = new Employee(1, "Ram");  
13 emp.display();
```

# Arrow functions(Arrow Function in Class)

- A class can include an arrow function as a property, see example18.ts

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example18.ts  
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example18.js  
1 Ram
```

# TypeScript - Class

- In object-oriented programming languages like Java and C#, classes are the fundamental entities used to create reusable components. Functionalities are passed down to classes and objects are created from classes. However, until ECMAScript 6 (also known as ECMAScript 2015), this was not the case with JavaScript. JavaScript has been primarily a functional programming language where inheritance is prototype-based. Functions are used to build reusable components. In ECMAScript 6, object-oriented class based approach was introduced. TypeScript introduced classes to avail the benefit of object-oriented techniques like encapsulation and abstraction. The class in TypeScript is compiled to plain JavaScript functions by the TypeScript compiler to work across platforms and browsers.

# TypeScript - Class

- A class can include the following:
  - Constructor
  - Properties
  - Methods
- See example19.ts for an example of a class in TypeScript

```
1  class Employee {  
2      empCode: number;  
3      empName: string;  
4  
5      constructor(code: number, name: string) {  
6          this.empName = name;  
7          this.empCode = code;  
8      }  
9  
10     getSalary(): number {  
11         return 10000;  
12     }  
13 }
```

# TypeScript - Class

- The TypeScript compiler will convert the above class to the following JavaScript code using closure:

```
JS example19.js ▸ ...
1  var Employee = /** @class */ (function () {
2      function Employee(code, name) {
3          this.empName = name;
4          this.empCode = code;
5      }
6      Employee.prototype.getSalary = function () {
7          return 10000;
8      };
9      return Employee;
10 }());
11
```

# TypeScript – Class(Constructor)

- The constructor is a special type of method which is called when creating an object. In TypeScript, the constructor method is always defined with the name “constructor”. In example20.ts, the Employee class includes a constructor with parameters empcode and name. In the constructor, members of the class can be accessed using this keyword. E.g. this.empCode or this.name. It is not necessary for a class to have a constructor.

```
example20.tsx ▸ Employee ▸ constructor
1  class Employee {
2      empCode: number;
3      empName: string;
4
5      constructor(empcode: number, name: string) {
6          this.empCode = empcode;
7          this.empName = name;
8      }
9  }
```

# TypeScript – Class(Constructor)

- Example20.ts

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc example20.ts  
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example20.js
```

- [C:\AlbertLam\Angular7-UT-Sept2019\Session2\Examples\example20.js]

dit View Search Document Project Tools Window Help

```
1 |var Employee = /** @class */ (function () {  
2 |    function Employee(empcode, name) {  
3 |        this.empCode = empcode;  
4 |        this.empName = name;  
5 |    }  
6 |    return Employee;  
7 |})();  
8 |
```

# TypeScript – Class without Constructor

- See example21.ts

rch Document Project Tools Window Help

C:\AlbertLam\Angular7-UT-Sept2019\Session2\Examples\example21.ts

```
1 class Employee {  
2     empCode: number;  
3     empName: string;  
4 }
```

C:\AlbertLam\Angular7-UT-Sept2019\Session2\Examples\example21.js

```
1 var Employee = /** @class */ (function () {  
2     function Employee() {  
3     }  
4     return Employee;  
5 }());  
6
```

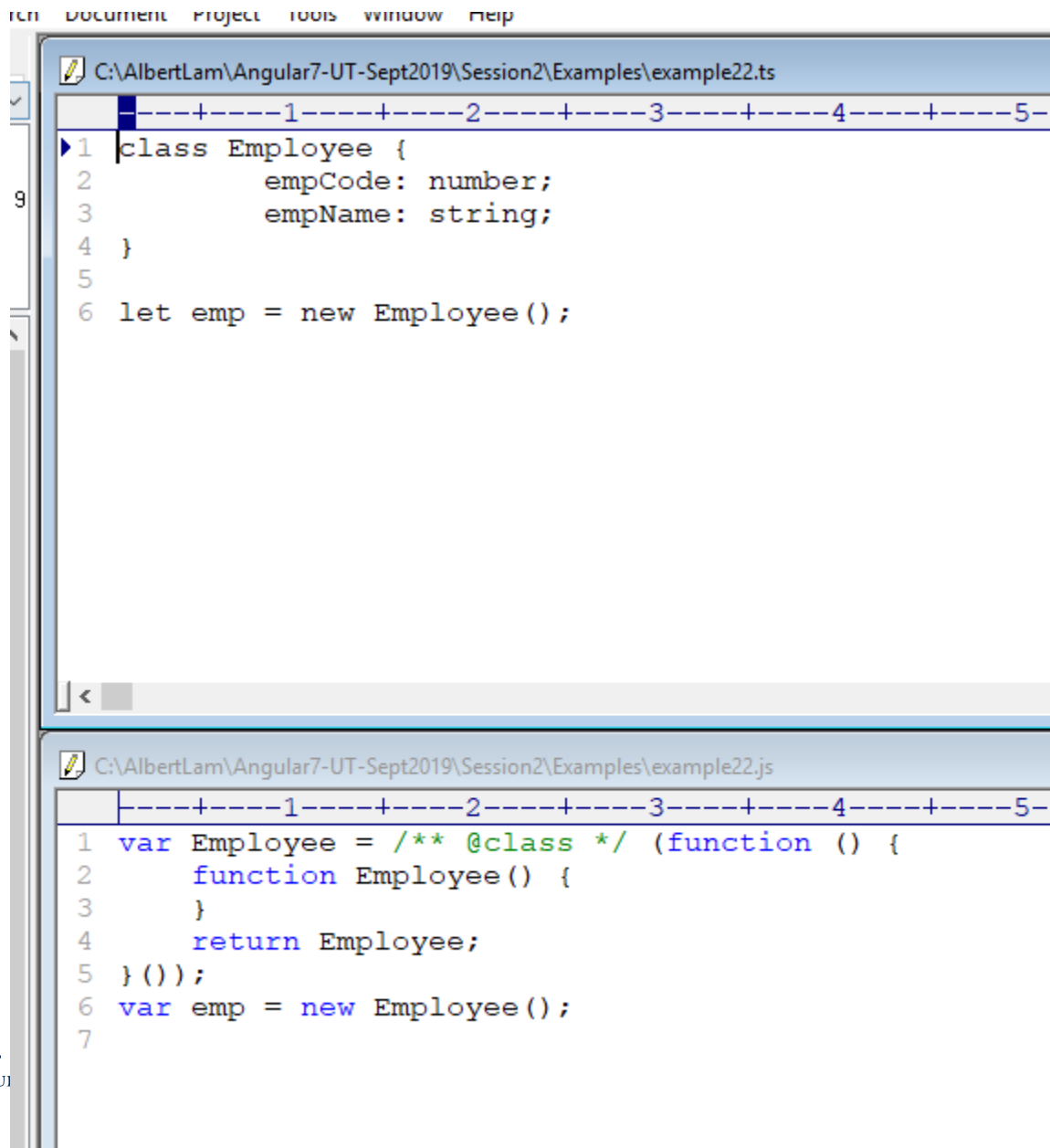


# TypeScript –Creating an object of class

- An object of the class can be created using the “new” keyword.
- Here, we create an object called emp of type Employee using "let emp = new Employee();". The class in example22.ts does not include any parameterized constructor so we cannot pass values while creating an object. If the class includes a parameterized constructor, then we can pass the values while creating the object.

```
TS example22.ts ▸ ...  
1  class Employee {  
2      empCode: number;  
3      empName: string;  
4  }  
5  
6  let emp = new Employee();
```

# TypeScript –Creating an object of class



The screenshot displays two code editors from an IDE. The top editor, titled 'C:\AlbertLam\Angular7-UT-Sept2019\Session2\Examples\example22.ts', contains TypeScript code. The bottom editor, titled 'C:\AlbertLam\Angular7-UT-Sept2019\Session2\Examples\example22.js', contains the corresponding JavaScript code. Both editors have a line number indicator on the left and a scrollbar on the right.

```
1 class Employee {  
2     empCode: number;  
3     empName: string;  
4 }  
5  
6 let emp = new Employee();
```

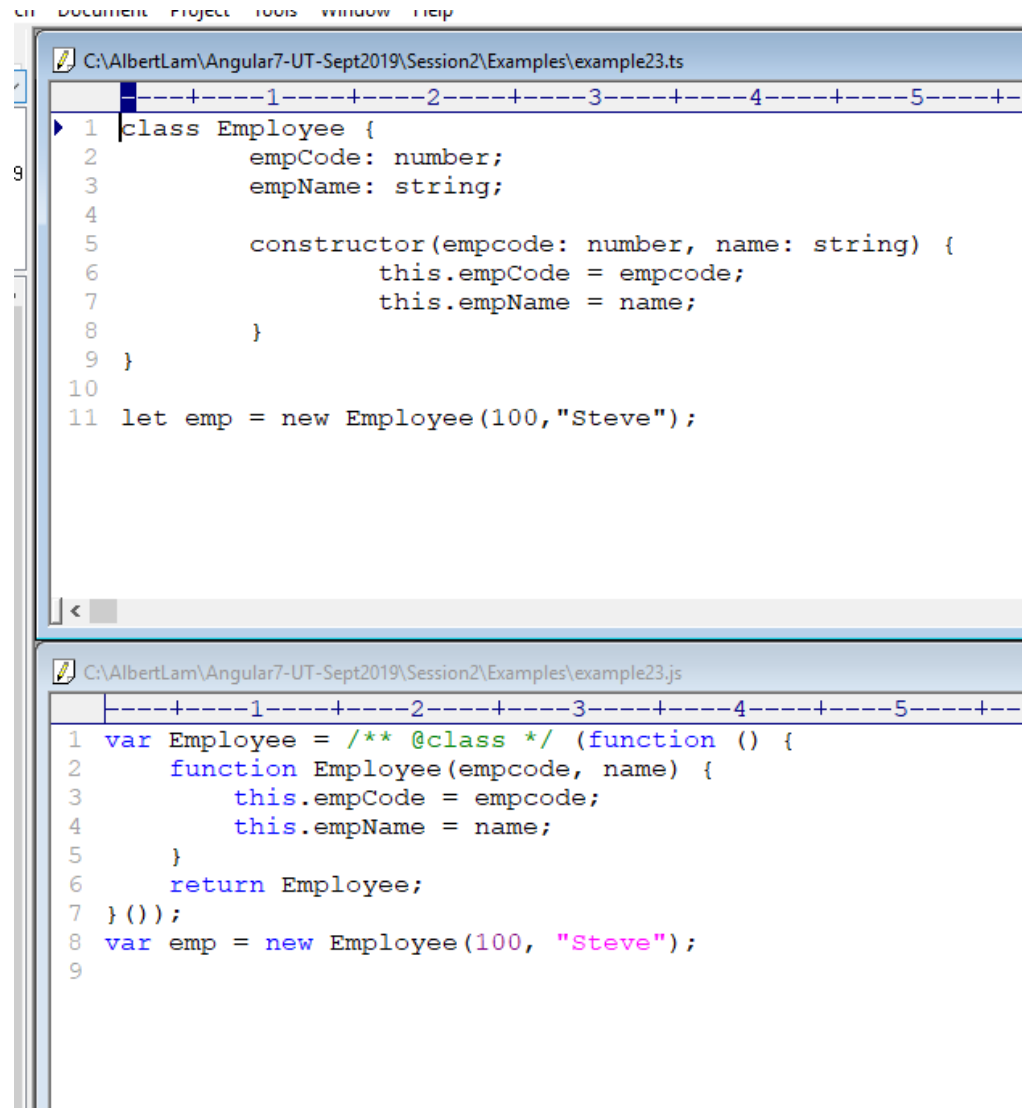
```
1 var Employee = /** @class */ (function () {  
2     function Employee() {  
3     }  
4     return Employee;  
5 } ());  
6 var emp = new Employee();  
7
```

# TypeScript –Creating an object of class

- In example23.ts, we pass values to the object to initialize the member variables. When we instantiate a new object, the class constructor is called with the values passed and the member variables empCode and empName are initialized with these values.

```
TS example23.ts ▸ Employee ▸ constructor
1  class Employee {
2      empCode: number;
3      empName: string;
4
5      constructor(empcode: number, name: string) {
6          this.empCode = empcode;
7          this.empName = name;
8      }
9  }
10
11  let emp = new Employee(100, "Steve");
```

# TypeScript –Creating an object of class



The screenshot displays two panels of code in an IDE. The top panel, titled 'C:\AlbertLam\Angular7-UT-Sept2019\Session2\Examples\example23.ts', contains TypeScript code for an 'Employee' class. The bottom panel, titled 'C:\AlbertLam\Angular7-UT-Sept2019\Session2\Examples\example23.js', shows the corresponding JavaScript code generated from the TypeScript file. Both panels include line numbers and a scrollbar on the right.

```
1 class Employee {
2     empCode: number;
3     empName: string;
4
5     constructor(empcode: number, name: string) {
6         this.empCode = empcode;
7         this.empName = name;
8     }
9 }
10
11 let emp = new Employee(100, "Steve");
```

```
1 var Employee = /** @class */ (function () {
2     function Employee(empcode, name) {
3         this.empCode = empcode;
4         this.empName = name;
5     }
6     return Employee;
7 }());
8 var emp = new Employee(100, "Steve");
9
```

# TypeScript –Inheritance

- Just like object-oriented languages such as Java and C Sharp, TypeScript classes can be extended to create new classes with inheritance, using the keyword `extends`. In `example24.ts`, the `Employee` class extends the `Person` class using `extends` keyword. This means that the `Employee` class now includes all the members of the `Person` class.
- The constructor of the `Employee` class initializes its own members as well as the parent class's properties using a special keyword `'super'`. The `super` keyword is used to call the parent constructor and passes the property values.

# TypeScript –Inheritance

- File:example24.ts:

```
TS example24.ts ▶ Person
1  class Person {
2      name: string;
3
4      constructor(name: string) {
5          this.name = name;
6      }
7
8  }
9
10 class Employee extends Person {
11     empCode: number;
12
13     constructor(empcode: number, name:string) {
14         super(name);
15         this.empCode = empcode;
16     }
17
18
19     displayName():void {
20         console.log("Name = " + this.name + ", Employee Code = " + this.empCode);
21     }
22 }
23
24 }
25
26 let emp = new Employee(100,"Bill");
27 emp.displayName(); //Name = Bill, Employee Code = 100
```

# TypeScript –Inheritance

- File: example24.ts:

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example24.js  
Name = Bill, Employee Code = 100
```

# Class Implements Interface

- A class can implement single or multiple interfaces. In example25.ts, the Employee class implements two interfaces – Iperson and Iemployee. So, an instance of the Employee class can be assigned to a variable of Iperson or IEmployee type. However, an object of type IEmployee cannot call the display() method because IEmployee does not include it. You can only use properties and methods specific to the object type.



# Class Implements Interface

- Example25.ts:

```
1  interface IPerson {
2      name: string;
3      display():void;
4  }
5
6  interface IEmployee {
7      empCode: number;
8  }
9
10 class Employee implements IPerson, IEmployee {
11     empCode: number;
12     name: string;
13
14     constructor(empcode: number, name:string) {
15         this.empCode = empcode;
16         this.name = name;
17     }
18
19     display(): void {
20         console.log("Name = " + this.name + ", Employee Code = " + this.empCode);
21     }
22 }
23
24 let per:IPerson = new Employee(100,"Bill");
25 per.display(); // Name = Bill, Employee Code = 100
26
27 let emp:IEmployee = new Employee(100,"Bill");
28 //emp.display(); //Compiler Error: Property 'display' does not exist on type 'IEmployee'
```

# Class Implements Interface

- Example25.ts:

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node example25.js  
Name = Bill, Employee Code = 100
```

# TypeScript-Export

- A module can be defined in a separate .ts file which can contain functions, variables, interfaces and classes. Use the prefix export with all the definitions you want to include a module and want to access from other modules.
- Employee.ts is a module which contains two variables and a class definition. The age variable and the Employee class are prefixed with the export keyword, whereas, companyName variable is not. Thus, Employee.ts is a module which exports the age variable and the Employee class to be used in other modules by import the Employee module using the import keyword. The companyName variable cannot be accessed outside the Employee module, as it is not exported.

# TypeScript-Export

- Employee.ts

TS Employee.ts ▸ ...

```
1  export let age : number = 20;
2  export class Employee {
3      empCode: number;
4      empName: string;
5      constructor(name: string, code: number) {
6          this.empName = name;
7          this.empCode = code;
8      }
9      displayEmployee() {
10         console.log("Employee Code: " + this.empCode + ", Employee Name: " + this.empName);
11     }
12 }
13
14
15 let companyName:string="XYZ";
```

# TypeScript-Import

- A module can be used in another module using an import statement.
- We exported a variable and a class in the Employee.ts. However, we can only import the export module which we are going to use. The following code only imports the Employee class from Employee.ts into another module in the EmployeeProcessor.ts file.s

TS EmployeeProcessor.ts ▸ ...

```
1 import { Employee } from "../Employee";  
2 let empObj = new Employee("Steve Jobs",1);  
3 empObj.displayEmployee(); // Output: Employee Code: 1, Employee Name: Steve Jobs
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc Employee.ts
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc EmployeeProcessor.ts
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node EmployeeProcessor.js
```

```
Employee Code: 1, Employee Name: Steve Jobs
```

# Importing the Entire Module into a Variable

- In EmployeeProcessor2.ts, we import all the exports in Employee module in a single variable called Emp. So we don't need to write an export statement for each individual module. In EmployeeProcessor2.ts, it will import age and Employee class into the Emp variable and can be accessed using emp.age and Emp.Employee.

TS EmployeeProcessor2.ts ▸ ...

```
1  import * as Emp from "../Employee"
2  console.log(Emp.age); //20
3
4  let empObj = new Emp.Employee("Bill Gates",2);
5  empObj.displayEmployee(); //Output: Employee Code: 2, Employee Name: Bill Gates
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc EmployeeProcessor2.ts
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node EmployeeProcessor2.js
20
Employee Code: 2, Employee Name: Bill Gates
```

# Renaming an Export from a Module.

- You can change the name of an export as shown in EmployeeProcessor3.ts. In EmployeeProcessor3.ts, the name of Employee export class is changed to Associate using {employee as Associate} . This is useful in assigning a more meaningful name to an export, as per your need which increases the readability.

TS EmployeeProcessor3.ts ▸ ...

```
1 import { Employee as Associate } from "../Employee"
2 let obj = new Associate("James Bond",3);
3 obj.displayEmployee(); //Output: Employee Code: 3, Employee Name: James Bond
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>tsc EmployeeProcessor3.ts
```

```
C:\workspace-UT-Angular8-JAN-2020\Session2\Examples>node EmployeeProcessor3.js
Employee Code: 3, Employee Name: James Bond
```