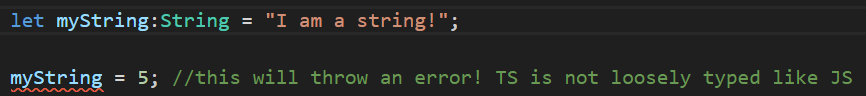
**What is TypeScript? (Started this at 10:29 EST)**

* TypeScript is the language we’ll use to provide functionality to our React front ends.
* **“TypeScript is a superset of JavaScript”**. (SuperSet - A **programming language** that contains all the features of a certain other language and has been expanded or enhanced to include other features as well.)
  + “TypeScript is like JavaScript, with **more strictness & OOP** principles built on”
  + **All valid JavaScript (JS) is valid TypeScript (TS)**. But the reverse is not always true...
  + TS must be **“transpiled”** into JS for a browser to read. Browsers do not have built-in support for TS.
    - React will transpile our TS for us :)
    - We can transpile a Typescript file from the command line using:
      * tsc myFile.ts
      * This will create the appropriate transpiled JS file
    - “Transpilation is a specific term for taking source code written in one language and transforming it into another language with a similar level of abstraction.”
    - So, while Java COMpiles into bytecode, TS TRANSpiles into JS.
  + TS is open source, object oriented, strongly typed, and mostly strictly typed.
    - Open source - basically just means free to use
    - OO - it supports classes, inheritance, interfaces
    - Strongly Typed - when you declare a variable, you need to declare a type as well.
    - Strictly Typed - when you declare a variable, you can’t change it’s type without some extra steps. No implicit type coercion!

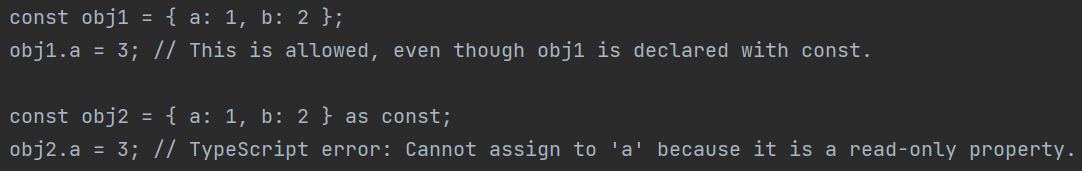


* + Just sounds like a mix of Java and JavaScript, right?
    - “It’s like JavaScript syntax with Java rules”

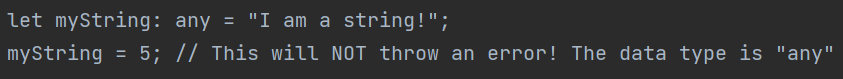
**TypeScript Data Types**

* **We tend to only use let or const.** var can be used but it’s less common.

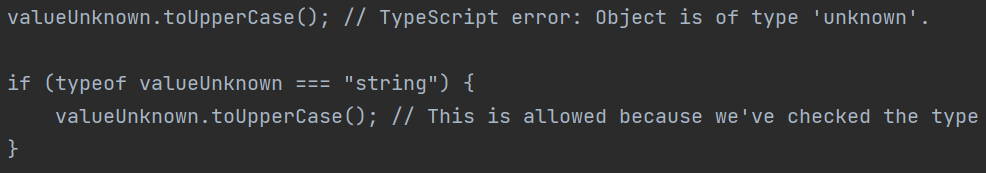
Note – we can declare variables with **“const”**, which makes the variable reference immutable, and **“as const”,** which makes the variable reference AND values immutable

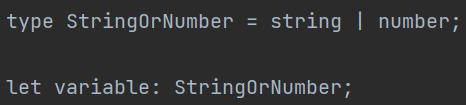
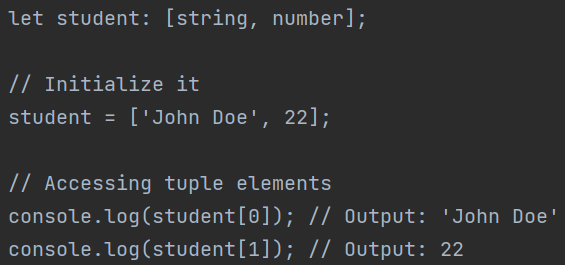
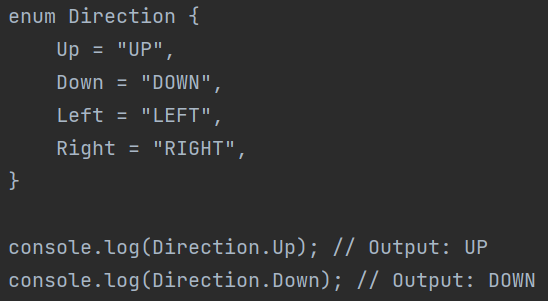
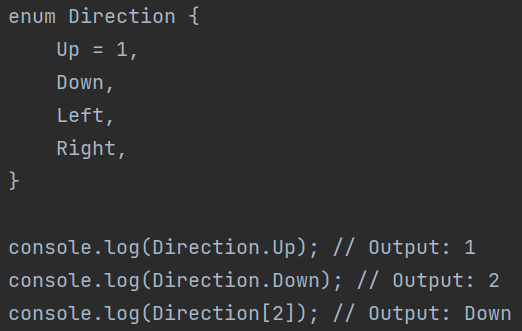
  
With just const, the values in this array can still change  
After adding “as const”, we can’t change the array’s values

* **TypeScript has all the basic datatypes of JS... Plus, way more:**  
  + **Any** - Because TS is transpiled into JS (which is loosely typed), sometimes we need the ability to declare a variable without a specific type.
    - It can be **“any” type it needs to be.** This makes our code more flexible!

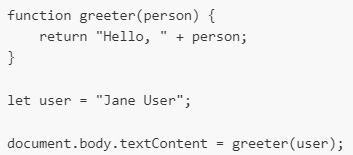


* + **Unknown –** Similar to “any”, but used when you **don’t know** the data type that may be used. This is rarely used.
    - The main difference is that you can’t perform operations on unknown types **until you’ve done a type check**

  
So unknowns can only be used within blocks that type check them first

* + **Arrays** - Arrays in TS are typed like Arrays in Java. So, only values of the declared Array type are acceptable, and they’re fixed in size.  
  + **Union types**: These allow you to specify that a value can be one of several types. You can use the | operator to combine multiple types into one.
    - 
    - 
    - 
    - 
  + **Tuples:** A tuple type is like an Array that has a **fixed number of elements** it contains, and **it specifies** **which types** it contains at specific positions.
    - They are a way to represent a record, or some collection of values that are related in some way. Like a very strict array
    - 
  + **Enums:** These allow us to define **a set of named consts**. Enums have some advantages over regular variables:
    - They make it easier to **document the intent** of a class
      * Gives a good idea of what the class does
    - Their values cannot change
    - They support type safety
  + There are two types we can use:
    - **String enums** store values as string literals
      * 
    - **Numeric enums** store values as numbers. By default, they start from 0 but you can set the value of the first item and the rest will increment from there.
      * 
  + **Never** & **Void:** These are **function return types**
    - **Void:** For a function that doesn’t return anything
    - **Never:** For a function will never complete because it “raises” (throws) an error.
      * You’d only really use this if you want to create a **custom error**.
        + The severity of the error is up to you! Do you want it to be handled? Or do you want it to crash your code?
      * It accomplishes the same thing as making a custom Exception in Java.
  + Functions can return any data type you want of course!

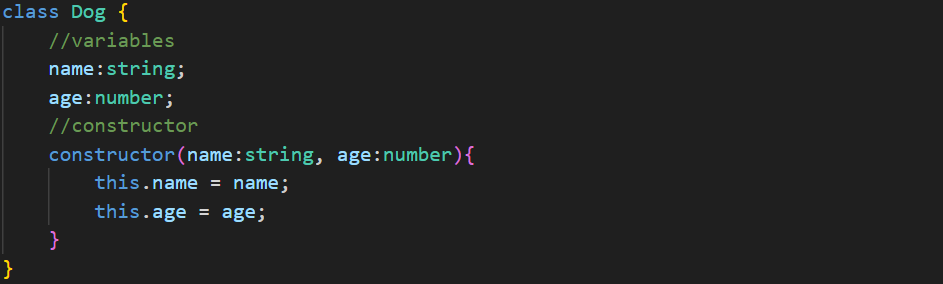
**Running a TypeScript file**

* **Disclaimer – we're not going to run TS outside of React. But here’s how you would do it:**
* First, you need Node.js installed... Ok good you already have it
* Install TypeScript using Node Package Manager (npm - included with node.js)
  + 
* Write a TS file!
  + 
* Run it from command line using the tsc command (TypeScript Compile) to transpile your TS into JS, that you can then run with node.
  + 

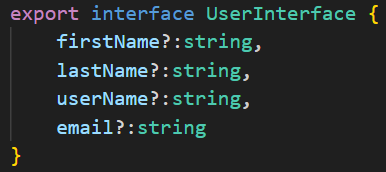
\*Probably start React right here\*

**TypeScript Classes**

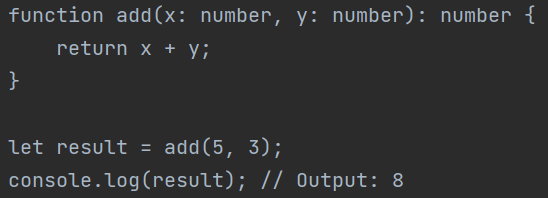
* TS is OOP - so it has support for Classes. It uses the “class” keyword to declare them, then the “new” keyword to create an object of that class.





* **TS has inheritance just like in Java** and uses the “extends” keyword in the same way.
  + We also have interfaces, which like in Java, define values and functions that inheriting classes must implement
  + 
  + And yes, we still use the “implements” keyword

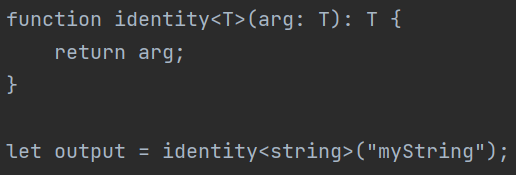
**TypeScript Functions**

* TS functions are like any other function we’ve seen in JavaScript, except we need to specify a return type.
* 

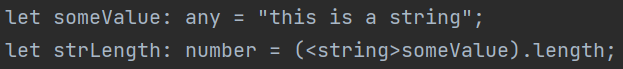
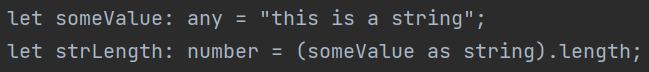
**TypeScript Decorators**

* Decorators are essentially the same as **annotations in Java.** They use the @ symbol and **provide additional functionality to your code.**
* Decorators can be attached to classes, functions, properties, parameters, etc.
  + We’ll see quite a few of these working with React

**Generics**

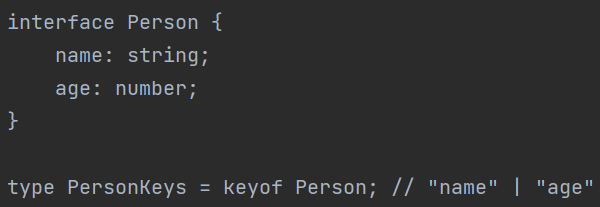
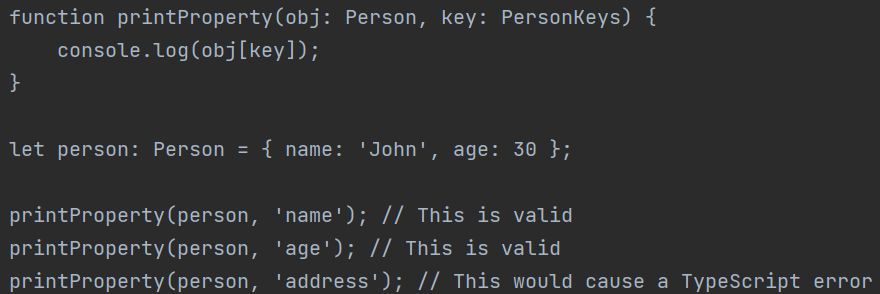
* Typescript has generics just like Java. They’re pretty much the same in syntax, behavior, and use case – they allow you to write reusable, flexible, and generalized types and functions.
  + 
    - In this example, we defined a function with a Generic type, allowing us to use any type of datatype when invoking the function
  + 
    - We can also declare Array Generics when the type of Array is known.

**Casting**

* Typescript has casting just like in Java, where we can explicitly change a variable’s datatype. Typescript lets you do it in two ways:
  + **Angle bracket syntax**
    - 
  + **“as” syntax** 
    - 

\*Probably get into day 2 of React here\*

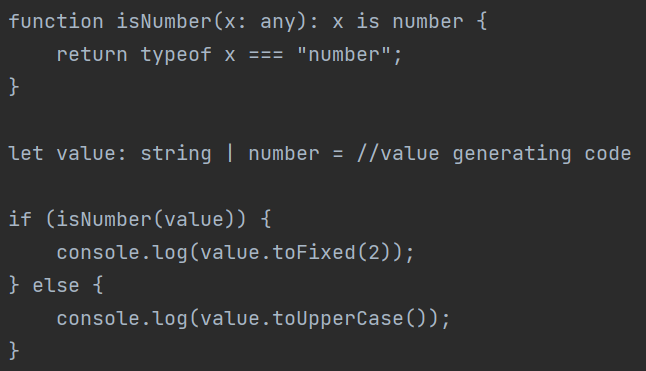
**Keyof**

* The keyof operator in TypeScript is used to create a union type representing the property names of a given type. It's a way to **store/query the keys of an object type.**
* 
* 
* In this example, we use “keyof” to create a new datatype called PersonKeys. It represents the keys of the Person interface, which are "name" and "age". This means that a variable of type PersonKeys can only be assigned the types of "name" or "age", both strings in this case.
* Then we use the created union type as a function parameter, limiting the value of that parameter to only values of string “name” or number “age”
  + Does that sound familiar? **Keyof creates union types based on an object/interface!**

**Utility Types**

* Utility types in TypeScript provide convenient ways to **transform types from one form to another.** They are essentially a set of generic types that aid you in certain type manipulations.
  + Some of the most used utility types are:
    - If you’re reading this,I realized this topic would be better to talk about next week stay tuned wahoo

**Type Guards**

* Type guards in TypeScript are a way to discern the datatype of a variable, before performing type-specific functionalities. They are a **design pattern** used by developers, not a built in TS construct.
  + What for?? Type guards are useful in scenarios where the type of a variable could be one of several types (a union type), and **you want to perform different operations depending on the actual type of the variable** at runtime.
    - We can use the **typeof** and **instanceof** keywords to make type guards.
*   
  In this example, we write a type guard for number type, generate a string OR a number from some code, and then use our type guard to determine datatype-based control flow
  + Note the ‘x is a number” return type – this is essentially a boolean but Ben will change this example to be more clear later
* Fun fact – this is how the “unknown” datatype works. You need to use a type guard before performing operations on an unknown.

**Typescript Configuration**

* TypeScript configurations are generally set in the aptly-named **tsconfig.json file.**
  + The tsconfig.json file is used to **specify the root files and the compiler options.** It's a crucial part of configuring your TypeScript projects
    - NOTE - we often don’t need to change the auto-generated configs at all.
* **Compiler options** are the many many many settings we can set within tsconfig.json
  + Here’s a good link for a reference on the config options in tsconfig <https://www.typescriptlang.org/tsconfig>

 example tsconfig.json

* If you remember only two compiler options, remember **strict and target**, because they’re on the curriculum lol:
  + **Strict:** This enables a wide range of **type checking behavior** that results in more type-safe programs. When strict is true, it enables all the options listed below (plus some more):
    - **noImplicitAny:** This raises an error on expressions and declarations with an implied any type.
      * as opposed to **explicitly** declared any types, which is fine.
    - **strictNullChecks:** This enables strict null checks – no nulls allowed in things like expressions.
    - **strictFunctionTypes:** This enables strict checking of function types.
    - **strictPropertyInitialization:** This ensures non-undefined class properties are initialized in a constructor.
    - **alwaysStrict:** This parses in strict mode and emits "use strict" for each source file.
  + **Target:** This specifies the **ECMAScript target version.** The default is ES5.
    - What is ECMAScript again? It’s basically the version of JavaScript (and by proxy, TypeScript) we’re using. It’s like Java versions.
  + **Module:** Sets the module system to be used. Setting it to “commonjs” will compile your TypeScript into standard JavaScript modules (transpilation!).
    - **React** will often have this set to “esnext”, AKA the latest version of ECMAscript
  + In the example above, you’ll also see **include and exclude** values. These are used to specify which files the TypeScript compiler should include or exclude **when it compiles your TypeScript code.**
    - Include contains the files/directories you want to be compiled
    - Exclude contains the files/directories you DON’T want to compile
* Also, do you remember the **tsc command** from the very beginning of our TS notes? The command we use to compile TS files? Well, it works by looking for **tsconfig.json** files!!
  + Each folder containing a tsconfig.json file will be treated as its own project by tsc, and be compiled according to the configs found in the tsconfig file.
  + **If no tsconfig.json is found, default configs will be used instead**