Chapter 1: Introduction to TypeScript

1.1 Introduction

- **TypeScript** is a JS superset, which means it is built up from JS and has more features and advantages. However, it cannot be executed by JS environments like browsers or NodeJS. Yet it is a powerful compiler which you run over your code to compile your TS code to JS, ie your end result is JS!
- TS compiles all the new features to JS "workarounds" that are more complex to write yourself.
- Most importantly, it adds **types** which will allow you an earlier opportunity of identifying errors in your code before the error occurs at runtime in the browser.
  - JS uses "dynamic types" resolved at runtime, while TS uses "static types" set during development. So can find bugs earlier.
- It also adds next-gen JS features that can be compiled down for older browsers (similar to how Babel makes JS compatible for older browsers).
- Also includes non-JS features like **Interfaces** or **Generics**. Also meta-programming features like **Decorators**.

Installation

- Since it uses npm, you need to have NodeJS first.
- Install globally with: **npm install -g typescript**
- Compile: **tsc helloworld.ts**
- You also want *npm init* and *npm install --save-dev lite-server* that will help you automatically restart your HTML pages when you make changes (instead of having to manually refresh everything). Remember to go to your package.json and under "scripts" insert a new "start": "lite-server". Now your page gets automatically reloaded whenever a file in the directory changes.
  - Learn what --save-dev does.

Brief Intro

- TS forces you to be clearer about your intentions and think about your code.
- Typing a **!** at the end tells TS that the code will never yield *null* and will always find an element. For eg:
  - const input1 = document.getElementById("num1")!;
  - If you aren't sure that the code will never yield *null*, then you should write an *if* statement to check if the element exists first or not.

1.2 Working with Types (see the 2 file)

number

- For both JS and TS, there is only the **number** type. There is no "integers" or "floats"; they would all be "numbers".
  - Do not misunderstand – all JS numbers are stored as double precision floating point numbers. **All numbers are floats**.

string

- All text values: 'Hi', "Hi", `Hi`.

boolean

- Both JS and TS have just "true" and "false"; DO NOT HAVE "truthy" or "falsy" values. For eg 0 is considered falsy (will fail the if-condition), but it is not related to data types – it is just JS and TS doing BTS work at runtime when they see it.

object

- For eg: {age:30}.
- When you hover over an object variable, it returns you an object-looking thing but is not actually an object (note no commas):
  - const person: {name: string; age: number;}
  - This is the **object type** inferred by TS. Object types are written almost like objects, but they do not have k-v pairs but instead have **key-type pairs**.

Let's say you have this JavaScript **object**:

```
const product = {
  id: 'abc1',
  price: 12.99,
  tags: ['great-offer', 'hot-and-new'],
  details: {
    title: 'Red Carpet',
    description: 'A great carpet - almost brand-new!'
  }
}
```

This would be the **type** of such an object:

```
{
  id: string;
  price: number;
  tags: string[];
  details: {
    title: string;
    description: string;
  }
}
```

So you have an object type in an object type so to say.

Array (capital A)

- Just like JS, an Array can store anything inside, eg [1, '2', false]. Hence, the type can be **flexible** or **strict** (regarding the element types).
- If flexible, can use **any[]**. But any[] means you lose the key reason why we use TS – to ensure correct types.
- If you do a loop on a string[], then can apply string actions because we know the elements are all strings.

ADDITIONAL CORE TYPES (that JS don't have and TS introduces)

Tuple

- Looks like [1,2] but it is a fixed length AND typed array.
- Have to define at the start.

Enum

- Pattern: enum {NEW, OLD}.
- Automatically enumerated global constant identifiers.
- It is a custom type.
- Useful for making a pointer system, eg 1 = ADMIN, 2 = PRO etc. Do it like this: *enum Role3 { NEW, ADMIN = 5, READ_ONLY = 99, AUTHOR = 'hello'}; // set your own*

Any

- Stores any kind of value with no specific type assignment.
- Can be: *any*, or *any[]* etc.
- While very flexible, disadvantageous because it takes away the point of TS. More used when you really cannot tell the type of data that will be coming in.

Others (see the 3 file)

- **Union** types using piping |. For eg, *input1: number | string*.
- But while your arguments can be flexible, your code may need to be adjusted such that each type is handled properly because TS cannot read union types at runtime.
- **Literal** types means your arguments can only be certain strings, for eg: *resultConversion: 'as-number' | 'as-text'*. Only allows these two options.
- You can create your own **type aliases / custom types** if you find yourself repeating things like *number | string* as these might be too long. Can also use it to create your own types.
  - Put at top: *type Combinable = number | string;*
  - Then use it *input1: Combinable.*
  - Another eg ***type User = {name: string; age: number};***
  - Then ***const u1: User = {name: 'Max', age: 30};***

For example, you can simplify this code:

```
function greet(user: { name: string; age: number }) {
  console.log('Hi, I am ' + user.name);
}

function isOlder(user: { name: string; age: number }, checkAge: number) {
  return checkAge > user.age;
}
```

To:

```
type User = { name: string; age: number };

function greet(user: User) {
  console.log('Hi, I am ' + user.name);
}

function isOlder(user: User, checkAge: number) {
  return checkAge > user.age;
}
```

Functions (see the 4 file)

- You can also set the return value of a function to be a certain type, for eg: *function add(n1: number)**: string** { … };*.
- However, it is best to let TS to infer the return type instead of explicitly setting it so.
- If you do not return anything in a function, then return type will be **void**. If you print(void), you will get **undefined**, which is also a type in TS (but how useful remains to be seen). For functions, the explicit casting of return results should be "void" if you are not returning anything. If you change it to "undefined", then you must at least have a "return;" statement at the bottom.
- You can set a variable as a **Function** type too to ensure that it will always be a function, ie ***let combineValues: Function;***.
  - But you can make them more specific so they can only be certain functions. The function below takes in two arguments that must be numbers, and returns a number.
  - *let combineValues: **(a: number, b: number) => number**;*
- Can also have custom function types, ie *type AddFn = (a: number, b: number) => number;*, and then *let add: AddFn;*.
- Can even have **callback functions** in the arguments.
  - Note that since the CB return value is void, so even if you do add a return statement in the CB, the result will still be void. JS and TS will ignore that return statement and just not return.
  - This means that even if you DO use a CB with a return statement, you are ALLOWED! Just that the return value will not be further used.

```
function addAndHandle(n1: number, n2: number, cb: (num: number) => void)
    const result = n1 + n2;
    cb(result);
}
addAndHandle(10, 230, (result) => { console.log(result);})
```

Even More Types (see 5 file)

- The type **unknown**, which works like *any* but has more restrictions. Best used when you don't know the type of the variable, but you have your checks later on to find its *typeof*, and deal it the appropriate workings.
  - *let userInput: unknown;*
  - Better than using *any* because of more restrictions on type. See the file for how to use it properly.
- The type **never** is used when throwing Errors. When your function is typed to throw errors, it does not really return void because when you print the return value, it is nothing as opposed to "undefined". Your throw statement will crash the script and never return.
  - Hence a more useful way to state that the function will never return anything instead of "void" or "undefined".
  - As a side note, an infinite while loop also returns never.

==========

Implementation

- function hello(n1: number, n2: boolean) { … }

Further Questions

- **Q: Why don't have the types when you declare at the const (explicit type conversion)?** This is because TS has a built-in feature called **(implicit) type inference**, where it can accurately infer what type the variable / constant will be when you declare it. It would be redundant to put it down there. The only exception is when you initialise without a value, for eg: *let num1: number;*.

Chapter 2: TS Compiler & Configuration

2.1 Configurations

Watch Mode

- Instead of needing to manually type "tsc abc.ts" every time you want to refresh, you can tell TS to **watch** the file and whenever that file changes, TS will recompile. Do that by:
  - **tsc app.ts -w**
  - **tsc app.ts --watch (both same results)**
  - The good thing is it focuses on changes on one file, but in larger projects this may not be the case.
- If you want TS to watch an entire directory of files, you can do the below instead:
  - **tsc --init**
  - Creates a **tsconfig.json** that tells TS that the folder and all subfolders should be managed by TS. You can edit this file for more configurations.
  - Subsequently, can manually recompile all TS files with: **tsc**
  - Alternatively, watch all TS files: **tsc -w**

2.2 tsconfig.json

include / exclude / files

- Can **exclude** files or folders from being watched by editing the tsconfig.json file:

```
    "skipLibCheck": true
  },
  "exclude": [
    "zzOld",
    "*wildcard.ts",
    "1 intro",
    "node_modules" // default excluded
  ],
  "include": [
    "app.ts",
    "analytics.ts"
  ],
  "files": [
    "app.ts"
  ]
```

  - By default node_modules is excluded so no need to add.
  - **include** works differently – if you have such a line, then any file that is NOT listed will NOT be recompiled.
  - Can also use **files** but it is pretty useless. It is similar to *include* just that can only specify files and not folders. Works the same.

compilerOptions

- target = the version of JS you want to compile to. Useful for compiling for certain (older) browsers.
- lib = option to specify which default objects and features TS notes. For example, when you have code such as "*document.querySelector("button")*", how does TS know that there is such a document object, that has a querySelector method, and a button exists? If *lib* is not set, then the defaults rely on the JS *target* option – it assumes that whatever is in the version of *target* would exist + assumes all DOM stuff exists.
  - However if you uncomment it / activate it, only things in it TS will understand. For eg, if you type "dom" in it, it will understand DOM elements.
  - The below is the same as if you had left it commented:
    - dom
    - esXXXX // latest version
    - dom.iterable
    - scripthost

sourceMap

- Helps with debugging and development.
- When you click on Inspector > Sources, you can see your decompiled JS code. But what if you want to see TS code instead for debugging purposes?
- When you set this to true, then compilation will generate extra **app.js.map** files which modern browsers will read and connect the JS files to the input TS files. Now in Sources the TS files will appear.
- With certain extensions, you can now edit directly in the Sources tab and it will translate into your VSC.

outDir and rootDir

- For bigger projects, you might want to organise your files. Your root directory should have a **src** folder containing all your TS files, and a **dist** folder having all the JS files.
- But doing so, the compiler will create the JS files beside the TS files.
- You can then set **outDir** to where the created files should be, ie "outDir": "./dist".
- You can also set "**rootDir**": "./src" so that TS will only compile files from this directory. Ignores all other TS files.
- The key difference for both compared to *include* and *exclude* is that the file structure is replicated from source to destination.

Others

- allowJs = if true, allow .js files to be compiled by TS.
- checkJs = if true, does not compile .js files but only checks for errors. Useful only if you want to make use of TS features but don't really care of the compilation, or to check extra JS files.
  - If you allow the above two options, then you must ensure you fill up the *include* and *exclude* options if not your JS files will have error / doubly checked (?).
- jsx for React. Not so sure what it does.
- declaration = if you need a .d.ts file to get a manifest of all the types that exist in your file.
- removeComments = removes comments from compiled JS files.
- noEmit = do not compile to JS files. Good for just checking files for errors and not waste time / resources creating the files.
- downlevelIteration = when you compile to older versions, for loops can rarely run into issues where not compiled correctly. This option if on gives you a more exact compilation. You might think then to always turn it on, but will produce more verbose code. Only use when you have loops and you see your loops behaving differently.
- noEmitOnError (default false) = if got error, the JS file still created. If true, then if any file has an error, ALL files (even those correct) will not be generated.

Strict Options

- strict = if true (default), enables all strict type-checking options, which is the same as setting all the below options in that paragraph as true.
- noImplicitAny (true default) = if false, it forces you to be clear the type of every argument / parameters you use. Cannot leave it to be implicitly guessed by TS. For variables still okay to be implicitly guessed (no errors raised) because TS can still guess what it may hold whereas parameters are external so TS may not know its type. (? Might be wrong but at least know it is to force you to set types more strictly).
- strictNullChecks = if true (default), tells TS to be strict with how you access and work with values that may potentially be null values, for eg when you querySelector a button that might not exist.
  - So what to do if TS raises an error? You can force an exclamation mark ! to tell TS you as a developer know the element exists.
  - If you are unsure if element exists, you can always wrap the code in an if-else conditional to only run if button exists.
- strictFunctionTypes = ? ignore for now.
- strictBindCallApply = useful when you work with *bind, call or apply*, which when true (default), checks if your method makes sense. Helps to make sure you don't use those methods incorrectly.
- strictPropertyInitialization =
- noImplicitThis = warns you if you use *this* keyword vaguely.
- alwaysStrict = ensures files generated always strict mode.

Additional Checks

- noUnusedLocals / noUnusedParameters = error if you created local variables / parameters that are not used.
- noImplicitReturns = error if a function sometimes returns something and sometimes does not. All functions must either not have a return statement, or return something in every scenario.
- noFallthroughCasesInSwitch = ensures you must have a break statement.

Chapter 3: Next-Generation JS and TS (revision)

3.1 Introduction

- This chapter is referring to modern JS that may not be compatible with older browsers: let, const, arrow functions, destructuring syntax, spread operator etc. TS will compile them such that they work in any case.
- Website for compatibility: https://kangax.github.io/compat-table/es6/

const / let

- Once **const** is declared, cannot change.
- **var** is more for older browsers and has a global and function scope (global variables can be used anywhere in script; variables in functions only available in that function). If a *var* is declared in an if-else loop, it is still global and can use outside of it. TS will raise an error but normal JS will not.
- Now we usually use **let** where it behaves like *var* except when in an if-else loop, it is local. This is because *let* introduces **block scope**, where the scope is limited to inside the curly braces, ie if (x) { let y = only_use_here }. This block scope is the same for functions, a for-loop etc, ie limited scope. This forces you to write cleaner code.

Arrow Functions

- const add = (a: number, b: number) => { a + b; };
- Single expression can ignore braces: const add = (a, b) => a + b;
- If you want to state types:
  - const printOutput: (a: number | string) => void = output => console.log(output);
- For buttons:
  - Button.addEventListener('click', event => console.log(event));

Default Values

- const add = (a: number, b: number **= 1**) => a + b;
- Note that default arguments should be the last parameter, because if not, how are you going to call the function?
  - Assume add = (a = 5, b). Then when you call, add(7) does not make sense. Hence defaults should be last parameters.

Spread Operators

- const hobbies = ['Sports', 'Cooking']
- const myHobbies = ['Walking']
- Remember in JS arrays are objects and objects are reference values. If you push, you change the memory but not the address.
  - This means when you const x = y, x and y are now pointers to the same address.
  - myHobbies.push(hobbies[0], hobbies[1]); // cumbersome
  - myHobbies.push(**...**hobbies);
- Spread tells JS to pull out all elements of that array and then treat each element individually, and usually add as new elements. Also works on objects.

Rest Operators

- If you expect a varying number of arguments being passed in, you can use the 3 dots again. It will merge all incoming parameters into an array, ie any[]. You can be more explicit by stating its type (see below).
- const add = (**...**numbers: number[]) => {
  - return numbers.reduce((curResult, curValue) => {
    - return curResult + curValue;}, 0);
  - };
- Where reduce(function on each element, starting value).
  - curResult is the current result.
  - curValue is for each element.
- Also works for tuples, ie number[number, number, number].

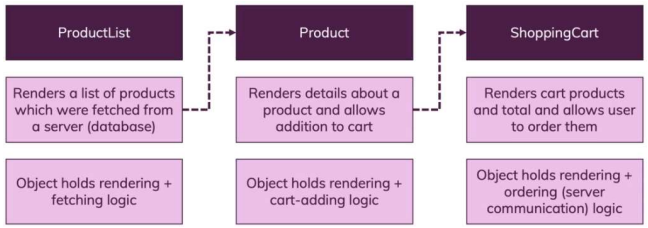
Destructuring

- NOTE: does not change your original array.
- const hobbies = ['Sports', 'Cooking'];
- const [hobby1, hobby2, …remainingHobbies] = hobbies;
- const person = { name: 'Max', age: 30 };
- const {name: username_I_wanna_use, age} = person;

3.2 Classes & Interfaces

Introduction

- In OOP, we want to simulate real-life entities in your code. For example a shopping website:



- **Classes** are blueprints for the objects (defines how objects should look like, what properties and methods they have etc), whereas **objects** are the things you work with in code (store data, execute methods). Objects are the **instances** of classes – you can quickly replicate multiple objects within the same structure.
- See file "1 basics on classes" for how to create classes and work with them generally.
  - Set types for properties.
  - Set parameters as *type: ClassName* so only instances of that class can use the methods.
- See file "2" for **access modifiers** such as the ***private*** keyword that makes properties / methods only accessible from inside the class.
  - WRONG (force not allowed): accounting.employees[2] = 'Janet'
  - CORRECT: accounting.addEmployee('Janet')
  - By default, all the other properties and methods are **public**. No need to state it explicitly.
  - Note that this is a NEW concept for JS. In the old JS, everything is public. Hence code that access "private" stuff may still work in the compiled JS. Also note that vanilla JS does NOT know either "public" or "private".
- Also see file "2" for **read-only modifiers** which means you cannot change certain things once initialised, for eg the ID of someone.
  - Also introduced only with TS.
- See file "3" for **inheritance** and about rewriting your new constructor function.
  - Use **super()** to call the constructor of the base class. The super() function must be the first thing in your new constructor before you can use the "this" keyword.
  - Despite a sub-class inheriting from its parent, it cannot access the parent's *private* properties / methods! To fix this, change *private* to ***protected***. A *protected* thing is available to all the classes that inherit from it.
  - NOTE: you only need to write a constructor if you want to pass in arguments when you instantiate the class with *new*.

Getters and Setters

- Available in JS and TS.
- A **getter** is a property where you execute a function / method when you retrieve a value, so you can add more complex logic. Use the **get** keyword.
  - Useful when you want to retrieve a *private* item from outside the object.
- Similarly, set the items with a **setter** using **set**.

Static Properties and Methods

- Static properties and methods allow you to add properties and methods to classes which are not accessed on an instance of the class, ie no need to call new ClassName() first, but which you can access directly on the class itself. ➔ **they are methods you can call directly on a class, not on an object created based on it**.
- Best for utility functions that you want to group or map to a class logically, or global constants which you want to always store in a class.
- Best examples = Math.PI or Math.pow().
- Just need to use the **static** keyword in front of that property or method.
- NOTE: You cannot access these static items inside the class itself with "this" from a non-static method, ie a non-static method cannot access a static item normally. Why? Because the "this" keyword refers to an instance of the class, yet whereas these static items do not have any instances created; the static items are detached from instances.
  - If you want to access them, eg ClassName.staticItem.
  - However, you CAN access static items inside a class with "this" using a static method. In fact, "this" and "ClassName" will both point to the same class in the same way (try and log it out!).

- Also note that "this" in JS/TS is different from that in Java. Do not confuse.

Abstract Classes

- If you know that a certain method should be available in all classes based on some base class (eg describe() in Department), but you also know that the exact implementation will depend on the specific version (describe() in IT vs Accounting), you want to FORCE your developers when they extend from the base classes to write a new version for each new sub-class, then you can add the **abstract** keyword to the base class's method. This forces all sub-classes that inherit from it to write their own methods with that name.
  - Must add the keyword at the top of the class definition, ie *abstract class Department { … }*.
  - Must also add it in front of the methods you want. You must then remove the curly brackets { … }, and write the return type.
- NOTE: Classes that are marked with *abstract* cannot be instantiated now. Can only do so for the sub-classes, who must now prescribe the properties or methods marked as *abstract*. ➔ **abstract classes cannot be instantiated but has to be extended**.

Singletons and Private Constructor

- In OOP, there is a pattern called the **singleton pattern** which means you will always only have exactly one instance of a certain class. Useful in scenarios where you somehow cannot use static methods or properties, or you do not want to. But at the same time, you want to ensure that you cannot create multiple objects based on a class; there is only exactly one object.
- For example, there can only be one Accounting Department.
- You need to add a *private* keyword in front of the constructor function for that class.
  - You will not be allowed to call *new* on that class anymore, ie cannot *new AccountingDepartment(…)*.
  - Hence you need to create static methods to get the instance. You also need to store a *private static* instance to refer to itself.

Interfaces (see folder 3 file 4)

- An **interface** describes the structure of an object, ie how it looks like. Only available in TS not JS, so when you compile there will be no mention of any interfaces.
  - Usage: *interface Person { … };*
- Interfaces are great to ensure that your class has certain features.
- **Q: Why do we have interfaces when we can just use custom types "type Person = { … };" which works as well?** While they are quite similar and can use interchangeably, there are some differences:
  - Interfaces can only be used to describe objects. Custom types can store other things like union types etc. Hence interfaces are clearer and that is why you often see them in real life.
  - You can implement interface in a class, ie an interface can be used as a **contract** (a contract forces an implementing class to have certain methods or properties) that a class can implement and adhere to.
    - This means a class can follow multiple interfaces, as opposed to inheritance where a class can only inherit from one parent only.
    - This also means interfaces are **used to share functionality amongst different classes**, not regarding their concrete implementation. It is useful to tell classes what structures they should have.
    - Similar to abstract classes in requiring those classes derived from it to be implemented in a certain way, just that for classes you have a mixture of requiring forced overwrites whereas in interfaces you do not type in any implementation at all (just the types).
    - Read here for more info on types vs interfaces.
- You can also add *readonly* for properties inside interfaces, which is useful to tell others that property can only be set once and thereafter only be read.
  - HOWEVER, a class that uses that interface with a *readonly* property DOES NOT inherit that *readonly* property for that item. The *readonly* property will only be applied to variables that are typed as that interface.
  - You CANNOT use *public* or *private*.
- You can also implement inheritance in interfaces. You do this by *extends* one interface with another, so any class that uses the child interface will need to have a structure of all interfaces as well.
  - Can combine multiple interfaces this way, ie *interface Greetable extends Named, AnotherInterface { … }*. Note how this is not possible for classes which can only inherit from 1 parent only.
- Can also tell your interface (or even class!) to have **optional properties** with ?:
  - outputName?: string; // properties
  - myMethod?(): void; // functions
  - constructor(n?: string) { … } // parameters
- Lastly, you can also make interfaces as function types to direct what types are the parameters and return value. **It is really identical to the custom function type method** (see folder 1 file 4), and you can just use custom types anyway, but in practice interfaces are more common. In earlier days, custom types did not have such features. Just know it exists. ➔ If you want to describe object structures, just use interfaces.

Chapter 4: Advanced Types

4.1 Intersection Types (folder 4 file 1)

- **Intersection types** allow us to combine with other types.
  - type ElevatedEmployee = Admin **&** Employee;
  - const e1: ElevatedEmployee = { … };
- Almost identical to interface inheritance, because you can just change the keywords "*type*" into "*interface*" and it would have still worked the same. **The key difference is interfaces is for object types whereas intersection types allow you to do with any other types**.
  - type Combinable = string | number; // union type
  - type Numeric = number | boolean;
  - type Universal = Combinable & Numeric;
- NOTE: slight different behaviour for objects vs union types ➔ for object types = combination of object properties (sort of like union of all properties); for union types = types they have in common (real intersection).
  - This anomaly is due to TS not really having a proper way to describe 'intersection'.
  - Intersection = variable must fulfil all of its component types. Union = variable can fulfil any (even just one) type.
  - Best way to think about it: is your variable of each type, ie is your variable a Combinable? Is it a Numeric? If yes for both, then it is of both types and is Universal.
  - Similarly, is your variable of type Admin? Is it of type Employee? If yes for both, then it is of type ElevatedEmployee.
- If an intersection fails because of no available types, then it is equivalent to type ***never***.
  - type a = number;
  - type b = string;
  - type c = number & string; // will always be type never.

4.2 Type Guards

- **Type guards** basically mean to check if a certain property or method exists before you try to do something.
- When you use union types (string | number), it gives you flexibility. However, this means you must do extra checks to ensure that the correct type is output at runtime.
- For example:
  - type C = string | number
  - function add(a: C, b: C) {
    if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
    }
    return a + b;
    }
  - Must do the *if* part if not TS will not allow due to vagueness.
  - The *if* part is actually a **type guard** (using ***typeof***) because it allows us to utilise the flexibility union types give and still ensure that our code runs correctly. This is because often you have functions that work with different types, but what exactly you do with the values depend on the types of the parameters.
- While ***typeof*** is able to check the type of the variable, it cannot differentiate amongst objects. For eg, you want only objects with the property *privileges* to have certain functions run on it, but *typeof* can only check if it is an object or not, ie *typeof emp === 'object'*. It cannot differentiate between the types Employee or Admin.
  - Hence instead use *if ('privileges' **in** emp') { … }* to check if a property belongs to an object.
  - Can also do the same way for methods, ie *if ('loadCargo' in vehicle)*.
- Alternatively, can use *if (vehicle **instanceof** Truck) {…}* for objects. While JS does not know what is "Truck" type, but it knows constructor functions, which classes ultimately are translated to constructor functions which JS is then able to find out if *vehicle* was created out of the *Truck* constructor function.
  - If you were to use *interface* instead of *class* here, then you cannot use *instanceof* because interfaces are not compiled to JS.
- In summary, type guards are: *typeof*, *in*, and *instanceof*.

4.3 Discriminated Unions

- This is a pattern you can use when working with union types that makes implementing type guards easier. More for object types.
- It has the same issue as above, which is solvable by the above solutions as well. However:
  - Relying on the *in* way may be difficult with large numbers of ifs.
  - Relying on *instanceof* is not possible if you use interfaces.
- Instead, we shall build a discriminated union by giving every interface (every object that is part of that union) an extra property.
  - Can use any name, usually *kind* or *type*. Its value will be the lowercase name of the interface.
  - Note that since we are dealing with interfaces, this is not actually a value of the *type* property; instead it is a **literal type**. We learnt previously that literal types means that the type can only be those string(s) only and nothing else.
  - Subsequently, you can deal with the *if* statements using a switch-case method.
  - Also great for avoiding spelling mistakes because the switch case only allows the possible types that are declared.

4.4 Typecasting

- Typecasting is to tell TS some value is of a specific type when TS is unable to detect on its own. For eg, if your code is to querySelector an element by id (which it won't know whether is it a button, a paragraph etc.).
- Typecasting does NOT convert one type into another. You are just telling TS that you know the type of a variable which TS cannot determine exactly, that's all. It will not help in any way at runtime.
- There are two ways to do typecasting and they are IDENTICAL.
  - Method 1: write the type in angled brackets in front, ie ***<HTMLInputElement>****document.getElementById('…')*. Note that your *tsconfig* file's *lib* property should have the *dom* item in the list so that TS will know the HTML elements.
  - Method 2: *document.getElementById(…) as **HTMLInputElement***. This is used to avoid clashing with React because of how React uses the angled brackets for the JSX.

4.5 Index Properties

- **Index types** is a feature that allows us to create objects which are more flexible regarding the properties they might hold.
- Useful where you know the value output type, but you cannot be sure in advance how many properties you will have and the names of these properties. For eg, an interface for errors.
- Start with square brackets, then either *key* or *prop*, and then the type of the value.
  - *interface ErrorContainer { **[prop: string]: string** };*
  - This means that the key/property is of type *string* (can only be strings / numbers / symbols, NOT boolean), and its value is of type *string*.
  - If a number is used as a key even though the type should be string, it will be read as string (no errors raised).
- You can add more properties inside, but they must be of the same type as the value type above.

4.6 Function Overloads

- Basically to call the same function with different parameters.
- Needed when your output type is undeterminable. For example, the add() function could either output a number or a string. But because we do not specify which, then we cannot do further functions on the output such as split() which only works on strings. Hence, we need function overloads to make things work.
  - One way of solving this is to typecast using *as string*. However, this is generally bad practice because of extra code.
- How to do:
  - On top of your function, write with the exact parameters types and return value type.

4.7 Optional Chaining

- Done when you're getting data from a backend / source where you cannot be certain if a property is defined. For eg, a fetched JSON sometimes may have a field, and sometimes may not.
- In plain JS, we would usually do an *if* statement and try to access those properties (ie *fetchedData.job && fetchedData.title)*.
- All you have to do for **optional chaining** is to add a **?** after your variable, ie ***fetchedUserData?.job.title***. This tells TS to check if each nested data exists before looking deeper into it. If it not exist (undefined), then TS will not throw a runtime error and instead just continue. Behind the scenes, it is compiled to an *if* check.

4.8 Nullish Coalescing

- Similar to optional chaining where you don't know if a certain data is a valid piece of data, ie null or undefined.
  - You may then want to store this unknown data into another variable. You may use a falsy comparison (*storedData = userInput || 'DEFAULT'*) but it will not work if userInput were an empty string ('') because it is falsy and will return 'DEFAULT' instead of just being empty.
- Instead, you can use the **nullish coalescing** with double question marks, ie *const storedData = userInput ?? 'DEFAULT'*. Hence if *userInput* is null or undefined (NOT an empty string or 0), then we will use the fallback aka *'DEFAULT'*. If it is not null or undefined, we will then use its value.

Chapter 5: Generics

5.1 Introduction to Generics

- Only in TS not JS. Concept is also in a few other programming languages.
- A **generic type** is a type which is kind of connected with some other type, and is really flexible regarding which exact type that other type is. For eg, a string[] is an Array type connected to the string type. Here, the type of values is what we are interested in (aka the string type in this eg) so we can tell TS exactly what type we are fixing on. Furthermore, knowing what type is inside that Array, you can then do further work on it, ie you know it is a string so you can .split().
- Anytime you see a notation like **Array<T>**, you are dealing with a generic type.
- One example is the **Array** (eg Array or any[]).
  - Do so with **Array<string>** which is same as **string[]**. Can also be Array<any> or Array<string | number> etc.
- Therefore, generic types causes you to provide additional information about the data type so you can do further processing. In fact, generics come in very handy in cases where you have a type that actually works together with multiple other possible types, eg an object which emits data of different types. Generics will help you create data structures that work together or wrap values of a broad variety of types (eg an array that can hold any type of data).

- Another example is the **Promises** type.
  - const promise: **Promise<string>** = new Promise(…);
  - Telling TS that this promise eventually returns a string.
  - Useful because when you .then(data) it, you can work with that data as if it were strings. If not, not knowing the type, TS may see as an error.

5.2 Creating our Generic Function

- In the same vein as above, when you write functions, for eg merging two objects, TS will know the output will be an object. But TS will not know this final object will have what properties, ie you cannot do further processing with it.
  - You can try solving with typecasting the final output, but it is very cumbersome to do that.
- Using **generic functions** is telling TS that while T and U are still objects of any types, but the final output will definitely be T&U. In contrast, having parameters as just "object" is vague and so will the output.
- Furthermore, we are telling TS that the types of T and U are dynamic to be discovered at runtime.
- You can also introduce **constraints** to ensure that T and U must be objects, because they can be numbers / strings which the code will still run but fail (without error!).
  - function merge<T **extends object**, U **extends object**>(…);
  - You can extends any type, even unions or Promises etc.
- In summary, it is to tell TS the input types of a function so that we can better work with the result of the function. It allows you to work with data in a TS best practices way.

keyof Constraint for Objects

- When you want to access an object using a key, TS may give an error because it cannot be sure that the object will definitely have that key.
  - *function extractAndConvert(obj: object, key: string) {*
    *return 'Value ' + **obj[key]**;*
    }
  - The above shows an error for obj[key] because TS cannot confirm that the object has that key property.
- Instead we can do the below:
  *function extractAndConvert<T extends object, U **extends keyof** T>(obj: T, key: U) {*
  return 'Value ' + obj[key];
  } // tells TS that U must be a key from the obj T
- BONUS: generic functions can almost replace the usage of type guards. Type guards will still be good if there are only a few concrete types you want to support in a function though.

5.3 Generic Classes

- Good for creating template classes that could be of any types. At the start, you have to choose which type T will be, and then the rest of the methods will only accept arguments of that type.
- class DataStorage<**T extends string | number | boolean**> {

```
private data: T[] = [];

addItem(item: T) {
  this.data.push(item);
}

removeItem(item: T) {
  if (this.data.indexOf(item) === -1) {
    return; // dont do anything if exact same object not passed
  }
  this.data.splice(this.data.indexOf(item),1);
}

getItems() {
  return [...this.data];
}
```

}

- However, it is **NOT** a good structure for objects. This is because in JS, the object argument passed in .removeItem() has a different memory address than the objects stored. Hence indexOf returns a -1, which would mean the last element instead. But this is not what we want. Hence, we need another implementation for objects.
  - It is still a good structure for primitive types like strings / numbers / boolean though.
- Can also use constraints inside the methods, eg *addItem<U>(item: U) {…}*. Done when you need a generic type inside a method but not the entire class. Not shown.
- In summary, generic types give you full flexibility of especially primitive types (ie you can do further processing with the result of a generic function/class), and also type safety.

5.4 Generic Utility Types

- This is about the built-in utility types from TS that give us more functionalities.
- Note that they are only in TS; compiles to nothing in JS. They just give you more strictness and extra checks.
- There are way more here. They are all generic because what they do is they take some other value of any type (they don't care the input types) and do something with it.

Partial

- **Partial<T>** tells TS that the object in the end will be type T, even if at the start it is not. Under the hood, Partial<> turns CourseGoal's properties as all optional, which is why we can initialise as empty obj at first, and then later add things step by step.
  - For the return statement, need the "*as*", because courseGoal is still Partial<CourseGoal> so we wanna typecast as a properfull CourseGoal.
- *function createCourseGoal(title: string, description: string, date: Date): CourseGoal {*
  *let courseGoal: **Partial<CourseGoal>** = {};*
  courseGoal.title = title;
  courseGoal.description = description;
  courseGoal.completeUntil = date;
  return courseGoal as CourseGoal;
  }
- .In summary, Partial<> is used when you want to temporarily switch one of your object types / interfaces to have its properties temporarily optional.

Readonly<>

- Tells TS that the RHS is an array of strings that are read only. This means if you try to push or pop something, it'll error.
  - *const meals: **Readonly<string[]>** = ['Dinner', 'Breakfast'];*
- Not limited to arrays. Can also use on objects so you cannot edit / add / remove any of its properties.

Chapter 6: Decorators

6.1 Introduction to Decorators
•