//@ts-nocheck skip ts checking for the whole file

`//@ts-nocheck` is a special comment used in TypeScript to disable type checking for the entire file in which it appears. When this comment is included at the top of a TypeScript file, the TypeScript compiler will skip type checking for that file, which can be useful in scenarios where you need to temporarily bypass type errors, such as:

1. **Migrating JavaScript to TypeScript**: When transitioning a large JavaScript codebase to TypeScript, you might want to convert files incrementally. Using `//@ts-nocheck` can allow you to get the file running without being blocked by type errors, while you incrementally add type annotations and fix issues.
2. **Third-party Libraries**: Sometimes, you might use a third-party library that doesn't have TypeScript type definitions. In such cases, you can use `//@ts-nocheck` to suppress errors until you find or create appropriate type definitions.
3. **Quick Prototyping**: When quickly prototyping or experimenting with code, you might use `//@ts-nocheck` to avoid dealing with type errors, focusing instead on functionality.

Here's an example of how it might be used in a file:

typescript

```
//@ts-nocheck

const add = (a, b) => {
  return a + b;
};

add(1, '2'); // This would normally cause a type error, but type checking is disabled.
```

Declare global

The `declare global` statement is used in TypeScript to extend the global scope with new types, interfaces, or variables. It allows you to add custom declarations to the global namespace, making them available throughout your entire TypeScript project without needing to import them in every file.

Here's a basic example of how `declare global` works:

// global.d.ts

```
declare global {
  interface Window {
        myCustomProperty: string;
  }
}

// Now, you can use `window.myCustomProperty` anywhere in your project
window.myCustomProperty = "Hello, world!";
console.log(window.myCustomProperty); // Outputs: Hello, world!
```

In this example:

- The `declare global` block is used to augment the `Window` interface by adding a new property called `myCustomProperty`.
- This augmentation makes `myCustomProperty` available on the `window` object throughout the entire project.

Use Cases

1. **Augmenting Existing Types**: You can add properties or methods to existing global types (e.g., `Window`, `Document`, etc.).
2. **Global Variables**: Declare global variables that are accessible across the project.
3. **Library Augmentation**: When you are using third-party libraries and want to extend their types or interfaces globally.

Key Points

- **Only for Declaration Files**: Typically, you use `declare global` in declaration files (e.g., `global.d.ts`).
- **Avoiding Name Clashes**: Be careful with naming to avoid clashes with existing global variables or types.
- **No Runtime Code**: It is purely for type declarations and does not generate any JavaScript code.

By using `declare global`, you can ensure that custom global typings are seamlessly integrated into your TypeScript project, improving type safety and development experience.

Declare global only augment, does not override the existing types

The `declare global` statement in TypeScript does not override existing types but instead augments or extends them. This means you can add new properties or methods to existing global types, but you cannot replace or modify the existing properties or methods directly.

Here's an example to illustrate how it works:

Augmenting Existing Types

If you want to add a new property to an existing global interface, like `Window`, you can do it using `declare global`:

```
// global.d.ts
declare global {
  interface Window {
        myCustomProperty: string;
  }
}

// Now, you can use `window.myCustomProperty` anywhere in your project
window.myCustomProperty = "Hello, world!";
console.log(window.myCustomProperty); // Outputs: Hello, world!
```

Example of What You Cannot Do

You cannot use `declare global` to directly change the type of an existing property. For example, you cannot change the type of `window.location` from `Location` to `string`:

```
// This is NOT allowed and will cause an error
declare global {
  interface Window {
        location: string; // Error: Duplicate identifier 'location'
  }
}
```

Correct Approach to Augmenting Types

To augment types correctly, you should only add new properties or methods, not modify existing ones:

```
// Correct way to add new properties
declare global {
  interface Window {
        myNewProperty: number;
        anotherProperty: () => void;
  }
}
```

```
// Using the new properties
window.myNewProperty = 123;
window.anotherProperty = () => {
  console.log("Hello from anotherProperty");
};
```

Summary

- **Augmentation**: `declare global` allows you to augment existing global types by adding new properties or methods.
- **No Overrides**: You cannot override or change the types of existing properties or methods.
- **Type Safety**: TypeScript will enforce these rules to maintain type safety and prevent accidental type conflicts.

Why Use `$` in Variable Names?

The `$` character in variable names is often used for:

- **Conventions**: To follow certain naming conventions. For example, in frameworks like Angular, `$` is sometimes used to denote observables or special variables.
- **Avoiding Collisions**: To avoid name collisions with other variables or reserved keywords.
- **Readability**: To make it clear that the variable has a specific role or importance in the code.

Overall, the `$` character is just part of the variable name and has no special significance beyond the naming convention chosen by the developer.

The $ symbol is often used in variable names in various programming languages for several reasons. While it doesn't have intrinsic meaning in most languages, it serves specific purposes depending on the context and conventions of the project or framework being used. Here are some common reasons:

Summary

The `$` symbol in variable names is mainly a convention used to:

- Follow framework-specific guidelines.
- Indicate special or significant variables.
- Avoid name collisions.

- Improve code readability and provide context.

It's a flexible convention that developers use to convey additional information about the purpose and usage of variables within their code.

1. Framework Conventions

Many frameworks and libraries use `$` as a naming convention to denote special variables, functions, or objects.

- **jQuery**: The `$` symbol is used as a shorthand for the `jQuery` function.

```javascript
const element = $('#myElement');
```

- **Angular**: AngularJS used `$` to denote core services and methods (e.g., `$scope`, `$http`).
- **RxJS**: It's common to suffix observables with `$` to distinguish them from regular variables.

```typescript
const data$ = this.http.get('/api/data');
```

2. Special Variables

The `$` symbol can be used to indicate that a variable has a special role or purpose, such as:

- **Observables**: In reactive programming with libraries like RxJS, appending `$` to variable names is a common convention to denote that the variable is an observable.

```typescript
const user$ = this.userService.getUser();
```

- **Temporary or Private Variables**: It can also be used to indicate that a variable is temporary or meant for internal use within a function or module.

```javascript
const $temp = computeTemporaryVal();
```

3. Avoiding Name Collisions

Using `$` can help avoid name collisions with other variables or reserved keywords.

```javascript
const $data = fetchData();
const data = processData($data);
```

4. Readability and Context

In some codebases, `$` is used to improve readability by providing additional context about the variable's usage or significance.

- **State Management**: In some state management patterns, `$` is used to denote state variables or store properties.

```javascript
const $state = { loggedIn: false };
```

5. Historical and Language-Specific Uses

- **PHP**: Variables in PHP are prefixed with `$` by language syntax.

```php
$variable = "value";
```

- **Perl**: Similar to PHP, variables in Perl also start with `$`.

Example in a TypeScript Context

In TypeScript, using `$` in variable names follows the same conventions mentioned above.

```typescript
interface UserData {
  id: number;
  name: string;
}

const userData$: UserData = { id: 1, name: "Alice" };

console.log(userData$.name); // Outputs: Alice
```

Declare global help to export var to global

```
import {
MauiRenderEngineRef,
UserState,
DataState,
AppState,
RawDataState,
FieldArray,
InvocableAction,
} from "@oci-maui/render-engine-types";

declare global {
// eslint-disable-next-line @typescript-eslint/no-explicit-any
function mauiExpression(...args: any[]): any;
const t: MauiRenderEngineRef["tFn"];
const $data: DataState;
const $user: UserState;
const $app: AppState;
const $rawData: RawDataState;
const fieldArray: FieldArray;
const invokeAction: InvocableAction;
}
```

What is ts utility types

https://www.typescriptlang.org/docs/handbook/utility-types.html

How to modify a key under typescript object

```
interface MyInterface {
 key1: string;
 key2: number;
}

// New interface extending the original interface
type ModifiedInterface = Omit<MyInterface, "key1"> & {
 key1: number; // Replacing the type of key1 with number
```

```
};

// Usage
const obj: ModifiedInterface = {
 key1: 10, // Now key1 is a number
 key2: 20,
};
console.log(obj);
```

What is object.entries? Return an array of array [key, value] pair. Enumerable - can be iterated over, enumeration - a list

The **`Object.entries()`** static method returns an array of a given object's own enumerable string-keyed property key-value pairs.

The phrase "own enumerable string-keyed property key-value pairs" refers to a specific aspect of objects in programming, particularly in languages like JavaScript.

Let's break it down:

Own: This means that the property belongs directly to the object itself, not inherited from a prototype.

Enumerable: This refers to whether the property will be iterated over by certain operations like loops. Enumerable properties can be counted by a for...in loop, for instance.

String-keyed: It indicates that the property's key, or name, is a string. In JavaScript, object properties can be accessed using either dot notation (obj.property) or bracket notation (obj['property']). When using bracket notation, the property key is always a string.

Property key-value pairs: This is a general term referring to the structure of an object, where each property has a corresponding value.

So, putting it all together, "own enumerable string-keyed property key-value pairs" describes the specific properties directly belonging to an object, which can be iterated over, accessed using string keys, and each of which has an associated value.

Convert enum into an a map. And then use map.get() to get something later

Converting an Object to a Map

The `Map()` constructor accepts an iterable of `entries`. With `Object.entries`, you can easily convert from `Object` to `Map`:

```
js
const obj = { foo: "bar", baz: 42 };
const map = new Map(Object.entries(obj));
console.log(map); // Map(2) {"foo" => "bar", "baz" => 42}
```

Expert typescript

https://nlbsg.udemy.com/course/typescript-for-professionals/learn/lecture/21451836#overview

1 primitive type

```
//besides normal string boolean and number
const notDefined: undefined = undefined;
const notPresent: null = null;
const peta: symbol = Symbol("star");
const biggy: bigint = 24n;
```

2 instance type

```
const regexp: RegExp = new RegExp("ab+c");

//below is generics
const array: Array<number> = [1, 2, 3];

//set will remove any repeated number
const set = new Set([1, 2, 3]);

//initialize the class with t type
class Queue<T> {
 //define the array type with t
 private data: Array<T> = [];
 //have to push the same item type as t
 push(item: T) {
   this.data.push(item);
 }
 //when pop out, we poping out the same type t
 pop(): T | undefined {
   return this.data.shift();
 }
}

const queue: Queue<number> = new Queue();
queue.push("string"); //Error: cannot accept string as t type, because t is initalized as number
```

4. Tuple - fixed length array

Seldom use this

```
//tuple is fixed array length, similar to array in array
// java list is not fixed length array
type tupleType = [number, number];
const tuple: tupleType = [0, 0];
const tuple2: tupleType = [0, 0, 3]; //error more than 3 number
const tuple3: tupleType = ['string']; //error cannot acceept string
```

7. Spread operator for infinite no of arguments

Declared an alias type usu has capitalized letter

```
//spread arg takes in an infinite no of args
//values is an array of number, ...[arg1, arg2] = arg1, arg2
const sum = (...values: number[]) => {
 //when values goes into function, it becomes an array
 return values.reduce((prev, current) => {
   return prev + current;
 });
};

sum(1, 2);
sum(1, 2, 3);
```

8. Duck typing

```
// duck typing - if it walks like a duck, then it must be a duck
// having extra info is okay

type Point2d = {
 x: number;
 y: number;
};

type Point3d = {
 x: number;
 y: number;
 z: number;
};
```

```
let point2d: Point2d = {
 x: 2,
 y: 3,
};

let point3d: Point3d = {
 x: 2,
 y: 3,
 z: 4,
};

// extra info is okay
point2d = point3d;
const takePoint2d = (point: Point2d) => {
 //point3d is like duck, that has extra power, but it is still a duck
};
takePoint2d(point3d);

// missing info// error below
point3d = point2d;
const takePoint3d = (point: Point3d) => {
 //cannot take in a less behaving duck
};
takePoint3d(point2d);
```

9 classes

```
// private - you cannot access the var directly using obj, but has to use setter
// protected - you cannot access var directly, but your children class could access it

class Animal {
 // private name: string;
 protected name: string;

 constructor(name: string) {
   this.name = name;
 }

 setter(name: string) {
   if (name) {
```

```
    this.name = name;
  }
 }
 move(meter: number): void {
  console.log(`${this.name} moved ${meter} m`);
 }
}

const cat = new Animal("cat");
cat.move(1);
cat.name = "dog"; //Error, cannot access the private var directly
//so that we control how user wants to set the variable in f setter
cat.setter("dog");

class Bird extends Animal {
 fly(meter: number) {
  // this.name is accessible by children if the var is protected
  console.log(`${this.name} flies ${meter} m`);
 }
}
```

10. Generics

What user initialized to be

```
// generics not any type, but what user passed in the type
class Queue<T> {
 data: T[];
 push(item: T) {
  this.data.push(item);
 }
 pop(): T {
  return this.data.shift();
 }
}

const queue = new Queue<number>();
queue.push(1);
queue.push("string"); // tell us right away we do not accept string here

console.log(queue.pop().toPrecision(1));
console.log(queue.pop().toPrecision(1)); //runtime error
```

12. Known is tighter than any, need to do type checking before doing anything

```
/const exampleAny: any = 123;

const exampleUnkown: unknown = 123;

/**
* any allow u to bypass ts control
* u can access it using any chained of obj attribute
* or assign to variable to a well defined type
*/
exampleAny.anything.underthesky.trim();
const anyBoolean: boolean = exampleAny;

/**
* unknown is tighter than any
* u cannot access it using any chained of obj attribute
* or assign to variable to a well defined type
*/
exampleUnkown.trim(); //error
unknowBoolean: boolean = exampleUnkown; //error

//need to determine unknown's types because manipulation unknown type
if (typeof exampleUnkown === "string") {
 exampleUnkown.trim(); //exampleUnknown is a string inside
}

if (typeof exampleUnkown === "boolean") {
 const unknowBoolean: boolean = exampleUnkown; //error without type checking
}
```

13. Type assertion
Use it sparingly because u r telling ts u know what type it is (instead of letting ts check for u)
Better to use type checking

19. Type declaration

Node js

- Express

- Process is env var where u did not specify the var type in your code base

You can declare it, or download the external type packages e.g. npm i @types/express

```
//process already declared in package file process.d.ts (d is declaration)
declare const process: any;
console.log(process.env.user);

//oci loom already has these types downloaded. no need to configure it again
import fs from "fs";
fs.writeFileSync("hello.text", "hello world");
```

21. Use promise to delay time instead of chained time out
Still not very clear with promise

- Why need promise (pending, resolve, and reject)
    - It is a state not execute immediately
    - Can hang in the air to wait for api call to completed
    - Not like normal variable which is executed line by line right away

```
const delay = (ms: number) => new Promise((res) => setTimeout(res, ms));

const mainAsync = async () => {
 await delay(1000);
 console.log("1s");
 await delay(1000);
 console.log("2s");
 await delay(1000);
 console.log("3s");
};
```

24. It is read only and cannot be modified

```
// readonly var - u can read but not modify
type Point = {
 x: number;
 readonly y: number;
};

const point: Point = {
 x: 1,
 y: 2,
```

```
};

point.x = 3;
point.y = 2; //Error, attribute y is read only, cannot be modified
```

27. Use is an attribute in an object to check its type - neaty way of doing ts

```
// use is attribute in an object to check the type of an object

type Square = {
 size: number;
};

type Rectangle = {
 length: number;
 width: number;
};

type Shape = Square | Rectangle;

const area = (shape: Shape) => {
 //is the attribute in an object to check its type
 if ("size" in shape) {
   return shape.size * shape.size;
 }
 if ("length" in shape) {
   return shape.length * shape.width;
 }
};

area({ size: 3 });
area({ length: 2, width: 3 });
```

28. Discriminate union - differentiate the which is which type among the union

```
type Square = {
 kind: "square";
 size: number;
};

type Rectangle = {
 //we are defining attribute name in the type
 kind: "rectangle";
```

```
 length: number;
 width: number;
};

type Shape = Square | Rectangle;

const area = (shape: Shape) => {
 //we are checking which type it is in the union type
 if (shape.kind === "square") {
   return shape.size * shape.size;
 }
 if (shape.kind === "rectangle") {
   return shape.length * shape.width;
 }
};
```

31. Use “==null” to check if it is null or undefined
TypeScript fully understands the JavaScript version which is `something == null`.

TypeScript will correctly rule out both `null` and `undefined` with such checks.

```
console.log(undefined == null); //true
console.log(null == null); // true

console.log(false == null); //false
console.log(0 == null); //false
console.log("" == null); // false
```

32. Intersection is the area of type which has all types of its children

```
// intersection which has types of all children
type Point2D = {
 x: number;
 y: number;
};

//intersection area has types of both children
type Point3D = Point2D & {
 z: number;
};
```

```
const variable: Point3D = {
 x: 1,
 y: 2,
 z: 3,
};

// console.log(variable);

type Point3 = {
 z: number;
};

type typeAll =
 //you can write like this to improve readability
 & Point2D
 & Point3;
```

34. Non null assertion - put a ! behind a variable to tell ts that it is not a null
Point!.x

Tell ts that point is not null

```
type Person = {
 name: string;
 email?: string;
};

const sendEmail = (email: string) => {
 console.log("send email to", email);
};

const ensureContactable = (person: Person) => {
 if (person.email == null) throw new Error(`Person is not contactable`);
};

const contact = (person: Person) => {
 ensureContactable(person);
 sendEmail(person.email!); //non null assertion (not working in oci though)
};
```

35. Interfaces extends is similar to type intersection, where the type are the combination of the children elements

Type Point3D = Point2D & {z: number}

Interface Point2D{
	X: number,
	Y: number,
}

Interface Point3D extends Point2D {
	Z: number
}

Interface Point3D {
	Z: number
}
Interface Point3D **extends Point2D** {
	Z: number
}

38. Never type - some function that never returns or in a infinite loop
You can only assign never to a type never

```
//function that never returns that never
const failTest = (msg: string): never => {
 throw new Error(msg);
};

//function in infinite loop is never
const sing = (): never => {
 while (true) {
   console.log("keep sing");
 }
};
```

But you can use never end of line to check all cases are checked

```
type Square = {
 size: number;
};

type Rectangle = {
```

```
 length: number;
 width: number;
};

type Circle = {
 radius: number
}

type Shape = Square | Rectangle | Circle;

const area = (shape: Shape) => {
 if ("size" in shape) {
   return shape.size * shape.size;
 }
 if ("length" in shape) {
   return shape.length * shape.width;
 }
 if ("radius" in shape) {
   return 3.14 * shape.radius * shape.radius;
 }

 //if all shape has been checked, the other shape would be a never variable
 //you can use this technique to ensure all cases are checked
 //but not very practical to use
 const _ensureAllCasesAreHandled: never = shape;
};
```

40. Definite assignment declaration

TS think we did not assign a number to a variable at the point of declaration. So we need to tell ts that the a number has been assigned to be the variable

```
//we are telling that we definitely assign a value to the var with !
// that is not a null
let dice!: number;

function rollDice() {
 dice = Math.floor(Math.random() * 6) + 1;
}

rollDice();
```

```
console.log("the dice is ", dice); //might have error without !
```

41. User define type guard

We can create type guard ourselves

```
type Square = {
 size: number;
};

type Rectangle = {
 length: number;
 width: number;
 string: string
};

type Shape = Square | Rectangle;

//user define type guard
//"shape is Square" is type predicate=> return boolean on this condition
//without this, type guard does not work
const isSquare = (shape: Shape): shape is Square => {
 return "size" in shape;
};
const isRectangle = (shape: Shape): shape is Rectangle => {
 return "length" in shape;
};

const area = (shape: Shape) => {
 if (isSquare(shape)) {
   //shape is Square return type will narrow any block guarded by the call to the
function
   //so within shape must be a rectangle
   return shape.size * shape.size;
 }
 if (isRectangle(shape)) {
   console.log(shape.string.trim())
   return shape.length * shape.width;
 }

 const _ensure: never = shape;
 return _ensure;
};
```

```
area({ size: 3 });
area({ length: 2, width: 3 });
```

Why must we use shape is Square as return type?

```
//change return type to boolean
//type guard will not recognize whether it is square or rectangle
const isSquare = (shape: Shape): boolean => {
 return "size" in shape;
};

const area = (shape: Shape) => {
 if (isSquare(shape)) {
   return shape.size * shape.size; //error: size does not exist on shape
 }
};
```

Part of ts documentation

User-Defined Type Guards

It would be much better if once we performed the check, we could know the type of `pet` within each branch.

It just so happens that TypeScript has something called a *type guard*. A type guard is some expression that performs a runtime check that guarantees the type in some scope.

Using type predicates

To define a type guard, we simply need to define a function whose return type is a *type predicate*:

```
function isFish(pet: Fish | Bird): pet is Fish {
 return (pet as Fish).swim !== undefined;
```

```
}
```
Try

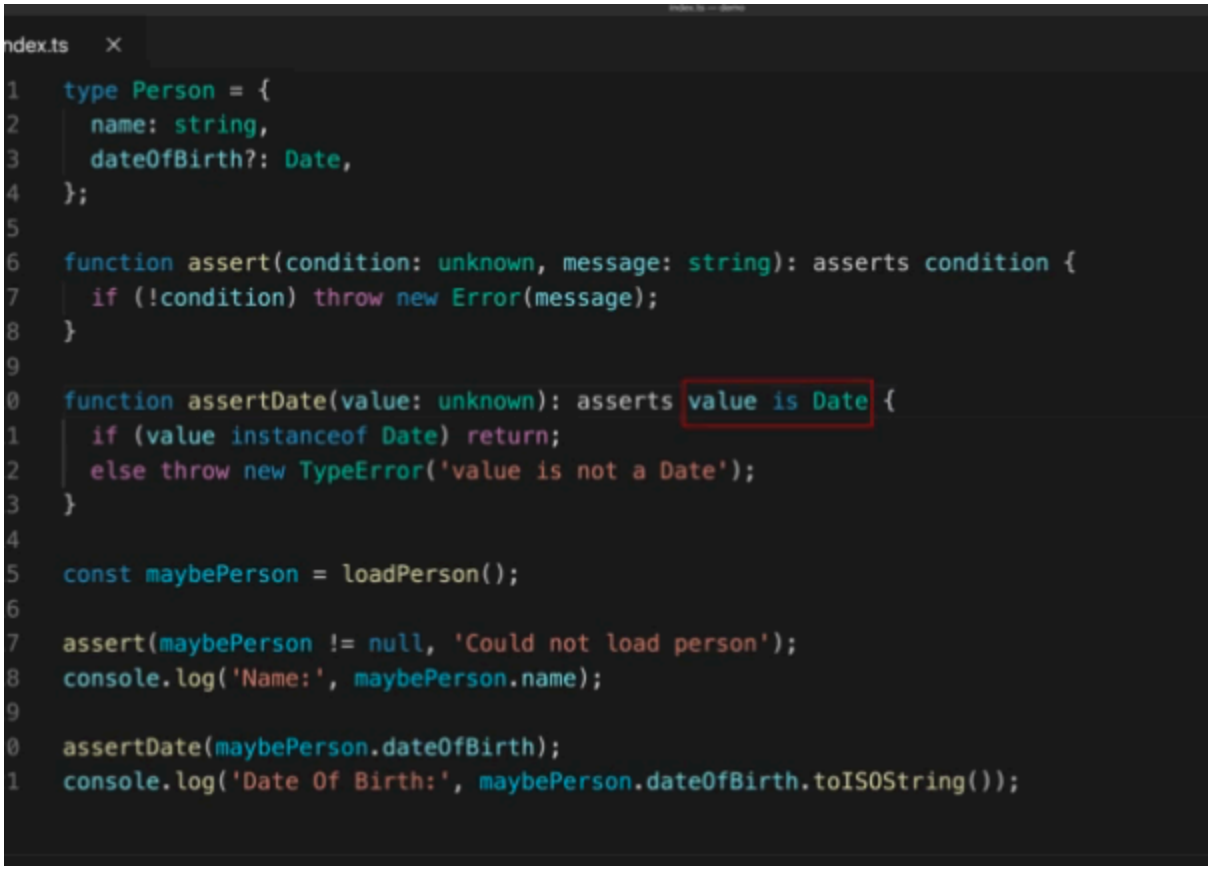
`pet is Fish` is our type predicate in this example. A predicate takes the form `parameterName is Type`, where `parameterName` must be the name of a parameter from the current function signature.

Any time `isFish` is called with some variable, TypeScript will *narrow* that variable to that specific type if the original type is compatible.

```
// Both calls to 'swim' and 'fly' are now okay.
let pet = getSmallPet();

if (isFish(pet)) {
 pet.swim();
} else {
 pet.fly();
}
```

42. Assertion function
You can create assertion function to check if a value is not undefined, before proceeding doing anything.
Usually it is written for application testing (not for development, because u do throw error)

Throw error, the execution will stop at the current line
If there is no error, then print out the name (however, ts is complaining mayBePerson.name still might be undefined. So we need to put the return value as assert type.



43. Use function overload from Java, you function could use 1 para, 2 para or other combination of paras

But not sure how const works for this
Rarely use in functional programming

```
function makeDate(timeStamp: number): Date;
function makeDate(year: number, month: number, day: number): Date;
function makeDate(timeStampOrYear: number, month?: number, day?: number): Date {
 if (month != null && day !== null) {
   return new Date(timeStampOrYear, month - 1, day);
 } else {
   return new Date(timeStampOrYear);
 }
}

const doomsDay = makeDate(2000, 1, 1); // 1 jan 2000
const epoch = makeDate(0); // 1 june 1970

const inValid = makeDate(2000, 1); // error not such function
```

44. Use call signatures/ or function type

Call signature means the function type

We can define function in different ways
Function overloading also can for const function below

You can narrowly define your call signature to accept only certain combination of paramters

```
type Add = {
 (a: number): number;
 // (a: number, b: number): number;
 (a: number, b: number, c: number): number;
 //optional name
 debugName?: string;
};

const add: Add = (a: number, b?: number, c?: number) => {
 //if c is not null, then it is C, else 0
 return a + b + (c != null ? c : 0);
};

add(1);
add(1,2); //error does not allow, because no such call signature defined
add(1,2,3);
add.debugName = "Additiona function";
```

45. Index signature - how you define key value pair in an object

Index signature is used to represent the type of object/dictionary when the values of the object are of consistent types. Syntax: { [key: KeyType] : ValueType }

```
type Person = {
 displayName: string;
 email: string;
};

type PersonDictionary = {
```

```
 //this is how you define a key type [key: type]
 [username: string]: Person | undefined;
};

const persons: PersonDictionary = {
 jane: {
   displayName: "Jane doe",
   email: "Jane@gmail.com",
 },
};

console.log(persons["jane"]);

persons["John"] = {
 displayName: "John",
 email: "John@gmail.com",
};

delete persons["John"];

//if there is no such key
const result = persons["missing"];
console.log(result, result.email); //undefine, error of undefined
```

47. Readony array and tuples how to limit mutation on the object
Not very useful in real life

```
//readonly can be applied to parameter of a function
//u can put there to prevent on the fly mutation of a function

const reverseSorted = (input: readonly number[]): number[] => {
 // return input.sort().reverse(); reaadonly cannot modify inplace
// u can only create a copy using slice(), then sort and reverse
 return input.slice().sort().reverse();
};

const start = [1, 2, 3];
const result = reverseSorted(start);

console.log(start);
console.log(result);
```

```ts
type Neat = readonly number[];
//below is how to write read only for generic type
type Long = ReadonlyArray<number>;
```

48. Double assertion “as unknown as Point3D”
You can use double assertion to force ts to trust you

index.ts

```ts
type Point2D = { x: number, y: number };
type Point3D = { x: number, y: number, z: number };
type Person = { name: string, email: string };

let point2: Point2D = { x: 0, y: 0 };
let point3: Point3D = { x: 10, y: 10, z: 10 };
let person: Person = { name: 'john', email: 'john@example.com' };

point2 = point3;
point3 = point2; // Error
point3 = point2 as Point3D; // Ok: I trust you

person = point3; // Error
point3 = person; // Error
point3 = person as Point3D; // Error: I don't trust you enough
point3 = person as unknown as Point3D; // Ok: I doubly trust you
```

49. Const assertion

“As const”

- Object attributes are mutable
- But u can assign const assertion to stop that
- Once you assign to it
- The object attributes became string literal type

String is not assignable to string literal type

When you start to use const, it converts the prevent type widening

- Confine the type to literal string

https://blog.logrocket.com/complete-guide-const-assertions-typescript/

```ts
const x = 'x'; // has the type 'x'
let y = 'x';   // has the type string, due to string widening to more general type
string
let y = 'x' as const; // y has type 'x'
```

```
const layout = (settings: { align: "left" | "right"; padding: number }) => {
 console.log(settings);
};

const example = {
 //string type is not assignable to literal string type
 align: "left", //error
 padding: 0,
};

layout(example);
```

```
//quite useful, if there is type incompatability


const layout = (settings: {
 align: 'left' | 'right',
 padding: number
}) => {
 console.log(settings);
}

const example = {
 //change align: string to align: 'left' => string literal type
 align: "left" as const,
 padding: 0
}

layout(example);
```

50. This parameter -> you can use it in the parameter to access it
Not very useful currently

index.ts

```ts
function double(this: { value: number }) {
  this.value = this.value * 2;
}

const valid = {
  value: 10,
  double,
};

valid.double();

console.log(valid.value); // 20

const invalid = {
  valve: 10,
  double,
};

invalid.double(); // Error
```

51. Generic constraints

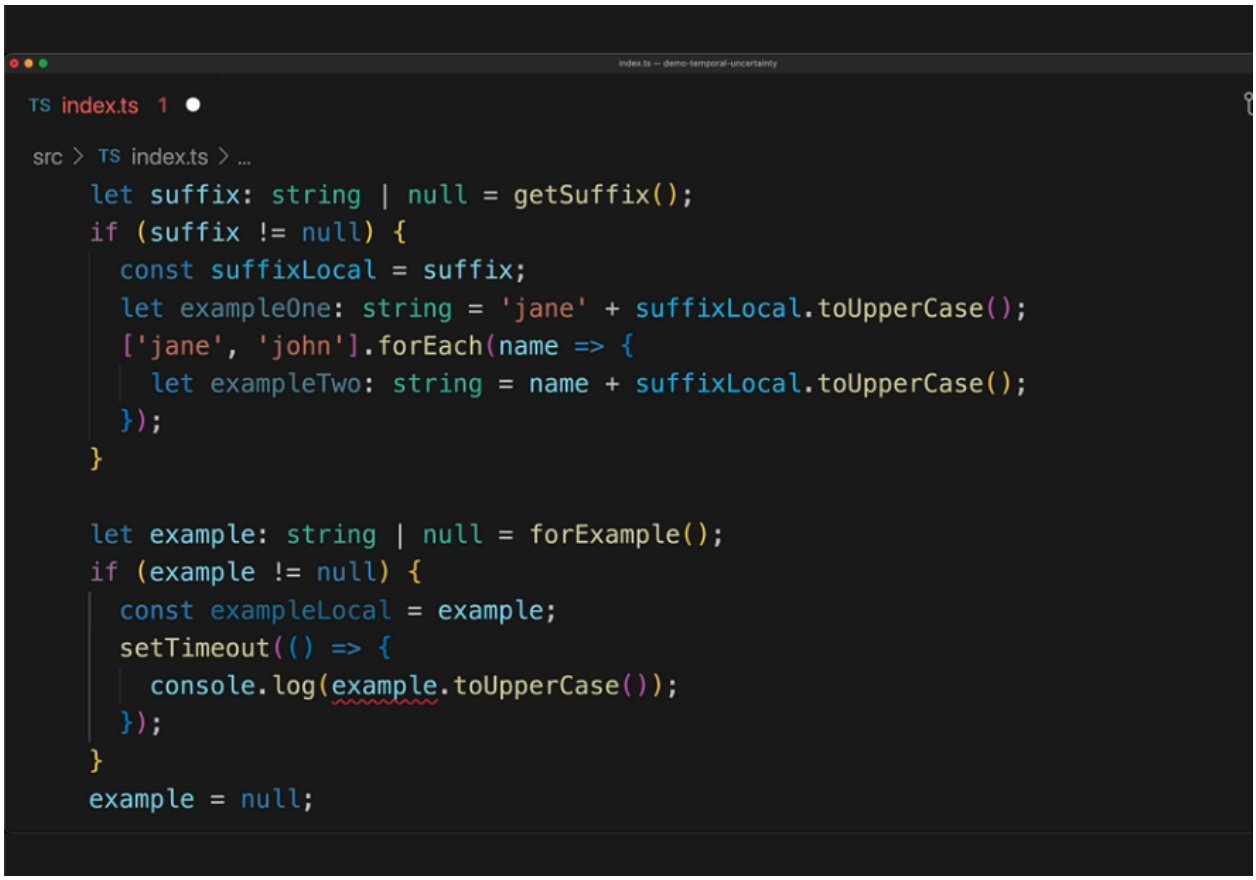
Use extend words to constraint the generic T, otherwise the type T is too wide

```ts
// extend keyword to constraint generic type
type NameFields = { firstName: string; lastName: string };
//T, define, parameter, return type
const addFullName = <T extends NameFields>(
 //T can be anything but must includes firstName and lastName properties
 obj: T,
 //return type is intersection of T and FullName (combined)
): T & { fullName: string } => {
 return {
   ...obj,
   fullName: `${obj.firstName} ${obj.lastName}`,
 };
};

const john = addFullName({
 email: "@vom",
 firstName: "joe",
 lastName: "sammy",
});

console.log(john.email);
```

```
console.log(john.fullName);
```

52. Dealing with temporal uncertainty

There async call where the var might be null.
So set a local variable to fix all these. (not sure why but not very practical)



53. Infer from the object the type using typeof

```
const unit = {
 x: 0,
 y: 0,
 z: 1,
};

//diffcult to write the type out again
//we can infer from the object
type unitType = typeof unit;

const point: unitType = {
 x: unit.x + 1,
 y: unit.y + 1,
 z: 1,
};
```

54Look up type

```
// look up the type in a nested large type
```

```
type apiResponse = {
 yourName: {
   persomal: {
     name: string;
     email: string;
   }[];
 };
 creditCard: {
   pmt: string;
 };
};
//similar to look up in array, we can look up type as well
type creditCardType = apiResponse["creditCard"];
//you can look up for nested type as well
//[0] below because we want to extract the object from the array
type personalType = apiResponse["yourName"]["persomal"][0];

const personalDetails: personalType = {
 name: "jerry",
 email: "vom",
};

const getPmt = (): apiResponse["creditCard"] => {
 return {
   pmt: "xxxx",
 };
};
```

55 Keyof extract the union types of an object keys

Play with Object and keyof object to ensure tighter control

```
// use typeof to get union types of the keys of an object type
type Person = {
 name: string;
 age: number;
};

const John: Person = {
 name: "John",
```

```
 age: 23,
};

//tighter constraint for obj and key
const getLog = (obj: Person, key: keyof Person): string | number => {
 const value = obj[key];
 console.log(key, obj[key]);
 return value; // return type is very general string | number
};

getLog(John, "name");
getLog(John, "email"); //log out error
```

Use generics and keyof to constraint the key and value of an object

```
// use typeof to get union types of the keys of an object type
type Person = {
 name: string;
 age: number;
};

const John: Person = {
 name: "John",
 age: 23,
};

// tighter constraint for obj and key
// generics in arrow f is together with the arg
const getLog = <Obj, Key extends keyof Obj>(obj: Obj, key: Key) => {
 const value = obj[key];
 console.log(key, obj[key]);
 return value; // infer return type as Obj[key], must be the value of the obj passed
in
};

getLog(John, "name");
// getLog(John, "email"); //log out error

const setLog = <Obj, Key extends keyof Obj>(
 obj: Obj,
 key: Key,
//below value is a lookup type return by ts
 value: Obj[Key],
) => {
 obj[key] = value;
};

setLog(John, "name", 23); //Error: 23 number is not string
```

Q extends Queries = typeof queries

Q extends the type of Queries, which by default is the typeOf quries

You can also create generic type from type itself
Q can be anything but must include basic properties of Queries (expand the functionalities)

```
export interface RenderOptions<Q extends Queries = typeof queries> {
 container?: HTMLElement
 baseElement?: HTMLElement
 hydrate?: boolean
 queries?: Q
 wrapper?: React.ComponentType
}
```

55. Conditional type

```
type IsNumber<T> = T extends number ? "number" : "other";

//= 'number'

type WithNumber = IsNumber<number>;

//= 'other'

type WithOther = IsNumber<string>;



export type TypeName<T> = T extends string

 //conditional type

 ? "string"

 : T extends number

 ? "number"

 : T extends boolean

 ? "boolean"

 : T extends undefined

 ? "undefined"

 : T extends symbol

 ? "symbol"
```

```
 : T extends bigint

 ? "bigint"

 : // : T extends Function

 // ? "function"

 T extends null

 ? "null"

 : "object";



//T is a type of t, whatever parameter passed in by user

// generic process, define, use as parameter, use as return type

const typeName = <T>(t: T): TypeName<T> => {

 //type casting

 if (t === null) return "null" as TypeName<T>;

 //typeof is a JS runtime function

 return typeof t as TypeName<T>;

};



//type is "string"

const str = typeName("hello world");
```

```
const num = typeName(123);


const bool = typeName(false);

const obj = typeName(null);




console.log(typeof null); // 'object'
```

57. U have string with never as union, the type return is always the string type

```
// auto verbose is string type below
//whenever u have union with never, it always return the type of not never
type Verbose = string | never;
//auto verbose is a number below
type VerboseNumber = number | never;

type Concise = string;

// below exclude null and undefined from T
//initialize T, define T
export type NoEmpty<T> = T extends null | undefined ? never : T;
//return string type
type Example = NoEmpty<string | null>;
//return type is string
type Expanded0 = NoEmpty<string> | NoEmpty<null>;
```

58. Infer is to use together with extends conditional type?, it can infer the member of an array or infer the return type of an array

- **It is just a to create a new type within the conditional type**

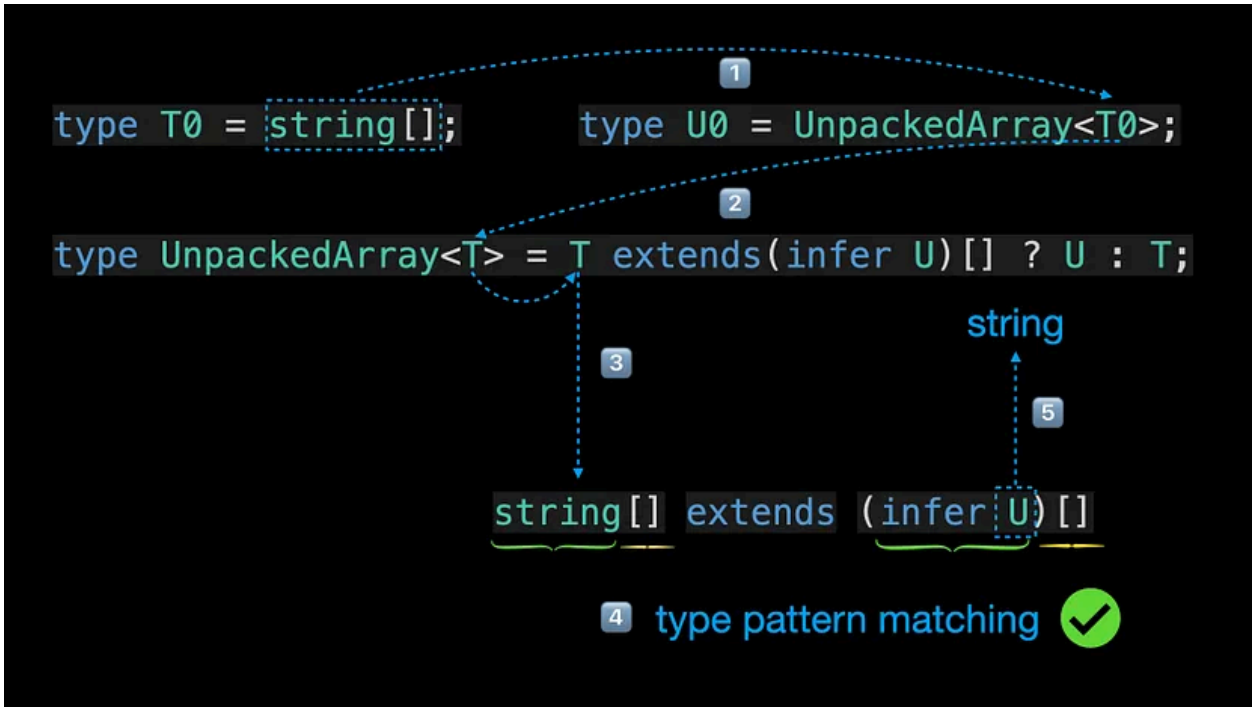
https://www.youtube.com/watch?v=hLZXJTm7TEk

Use ReturnType<typeof function> to infer the return type of a function, instead of writing it yourself

We can use infer member to infer the member type of an array

```
type IsArray<T> =
 //below infer the member element of the array and return its type
 T extends Array<infer member> ? member : "other";

type withArray = IsArray<string[]>;
type withNoArray = IsArray<string>;

//if an array, return its member type, else return its type
type UnboxArray<T> = T extends Array<infer member> ? member : T;

type UnboxStringArray = UnboxArray<string[]>; //string type
type AnythingElse = UnboxArray<string>; //string types
```

It should be noted that infer can only be used in the extends clause of the conditional type, and the type variable declared by infer is only available in the true branch of the conditional type.



ReturnType<typeof function> to infer the return type of a function,

```
export const createPerson = (firstName: string, lastName: string) => {
```

```
 return {
   firstName,
   lastName,
 };
};

//we do not need to write below anymore
type ReturnType<T> = T extends (...arg: any) => infer R ? R : never;
//first get the function type, then use ReturnType to extract the return type
type Person = ReturnType<typeof createPerson>;

const logPerson = (person: Person) => {
 console.log(person.firstName, person.lastName);
};
```

You can infer the type of the deep nested value of an object

55. Mapped type - You can loop through the object type to get its keys and value type

```
type Point = {
 x: number;
 y: number;
```

```
};

type ReadonlyPoint = {
 readonly x: number;
 readonly y: number;
};

//use mapped type to loop through each attribute of an array type
type MappedReadOnlyPoint = {
 readonly //item to loop through union keys "x" | "y" | "z"
//[item in Unions]: output
 [item in "x" | "y"]: number;
};

type MappedReadOnlyPoint2 = {
 readonly //use keyof to get the keys of the object
 //use type look up to get the value type
 [item in keyof Point]: Point[item];
};

type MappedReadOnlyPoint3<T> = {
 readonly //use keyof to get the keys of the object
 //use type look up to get the value type
 [item in keyof T]: T[item];
};

const center: MappedReadOnlyPoint3<Point> = {
 x: 2,
 y: 3,
};

// we do not need to write complicated MappedReadonlypoint3
//below ts has its built in function
const center2: Readonly<Point> = {
 x: 2,
 y: 3,
};

center.x = 4; // should error
center2.x = 4; // should error
```

60. Mapped type modifier

You can + and - all readonly or? Optional

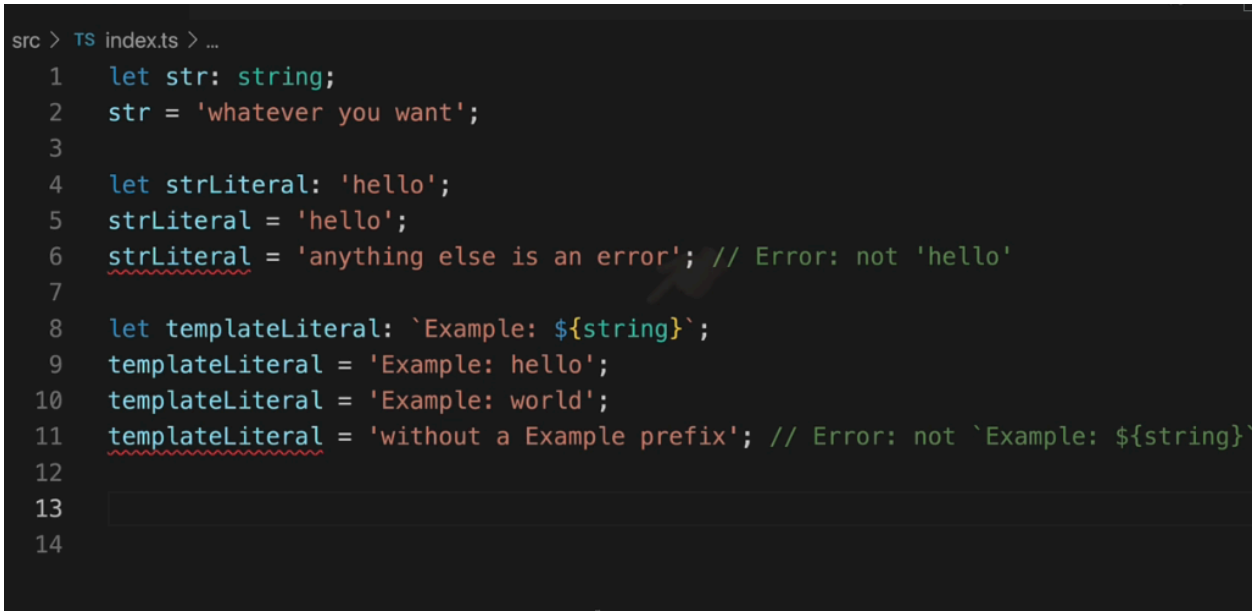
```
type Point = {
 readonly x: number;
 y?: number;
};

type Mapped<T> = {
 [P in keyof T]: T[P];
};

type Mapped2<T> = {
 +readonly //apply readonly to all (+ is just to make it readable)
 [P in keyof T]: T[P];
};

type Mapped3<T> = {
 -readonly //remove readonly to all
 [P in keyof T]: T[P];
};

type Mapped4<T> = {
 -readonly //remove readonly and optional to all
 [P in keyof T]-?: T[P];
};

type Result = Mapped4<Point>;
```

Use partial to make object attribute type optional

```
// use partial to get partial attributes of an object type

type Partial2<T> = {
 [P in keyof T]?: T[P];
};

class State<T> {
 constructor(public current: T) {}
 //Ts already has built in Partial, which makes all obj attributes optional
 update(next: Partial<T>) {
   this.current = { ...this.current, ...next };
 }
```

```
}

const state = new State({ x: 1, y: 2 });
state.update({ y: 33 }); //should have error, either full T or partial T
```

61. Template literal type is avail in new ts
But oracle ts does not have this yet



62. Use partial obj type to make all attributes optional

```
export type Partial<T> = {
 //make all attributes of obj optional
 [P in keyof T]?: T[P];
};

//we do not declare above because this is a built in func in ts
type Point = { x: number; y: number };

// same as {x?: number, y?: number}
type PartialPoint = Partial<Point>;

class State<T> {
 constructor(public current: T) {}
```

```
 update(next: Partial<T>) {
   this.current = { ...this.current, ...next };
 }
}

const state = new State({ x: 0, y: 0 });
state.update({ y: 34 }); //update current partially
console.log(state.current); //{x: 0, y: 34}
```

63. Required makes all obj attributes compulsory

```
// export type Required<T> = {
//   [P in keyof T]-?: T[P];
// };

type PartialPoint = {
 x?: number;
 y?: number;
};

// same as {x: number, y: number}
// required is a built in func
type RequiredPoint = Required<PartialPoint>;

class Rectangle {
 private config: Required<PartialPoint>;
 //when passing in parameter, it can be partial
 constructor(config: PartialPoint) {
   //this.config is required all attributes
   this.config = {
     //nullish operator, null or undefined, will always return values
     x: config.x ?? 0,
     y: config.y ?? 0,
   };
 }

 draw() {
   // no null checking is needed, always return a value
   console.log(this.config.x, this.config.y)
 }
}
```

65. Record<K, V> defines how an object and its internal element's shape

```
type Persons = Record<string, { name: string; role: string }>;

const persons: Persons = {};
persons["000"] = { name: "john", role: "admin" };
persons["10"] = { name: "john", role: "admin" };
persons["101"] = { name: "john" }; // error: missing role

//similar to Record
type PersonsVerbose = { [key: string]: { name: string; role: string } };

//if u specify the keys of Record, they must all be there

type PeopleWithRoles = Record<"admin" | "owner", string[]>;
const allPeople: PeopleWithRoles = {
 admin: ["J", "M"],
 owner: ["J", "M"],
};

const allPeople2: PeopleWithRoles = {
 admin: ["J", "M"],
 //Error: owner is missing
};
```

66. Use (string & {}), preventing primitive type taking over string literal type
so that primitive string will not take over the union string below
While still retaining the auto completion

Not very practical and useful

```
type StringAndObj = string & {};

//string type will not override the union type
//not working in OCI code base
type Padding = "small" | "medium" | (string & {});

const getPadding = (padding: Padding) => {
 if (padding === "small") return "12px";
 if (padding === "medium") return "16px";
 return padding;
```

```
};

const padding: Padding = "small";
//string did not override union type, autocompletion intact
const padding2: Padding = "9px";
```

70. Propertykey is a generic name used to rep keys in an obj

```
//typescript only accept string, number or symbol as the property key
const str: PropertyKey = "string";
const num: PropertyKey = 33;
const sym: PropertyKey = Symbol();
const obj: PropertyKey = {}; //Error cannot assign this

const valid = {
 [str]: "valid",
 [num]: "valid",
 [sym]: "valid",
 [obj]: "invalid", //invalid key
};
```

71. thisType is to avoid typing out the this again and again

```
//thisType is used to describe the type of this
//so that you do not have to write again and again
type MathCustom = {
 double(): void;
 half(): void;
};

export const math: MathCustom = {
 double(this: { value: number }) {
   this.value *= 2;
 },
 half(this: { value: number }) {
   this.value /= 2;
 },
};

const obj = {
 value: 1, //this type is an obj
```

```
 ...math,
};

obj.double();
console.log(obj.value);
```

```
type MathCustom = {
 double(): void;
 half(): void;
};
//thisType is used to describe the type of this
//so that you do not have to write again and again
export const math: MathCustom & ThisType<{ value: number }> = {
 double() {
   this.value *= 2;
 },
 half() {
   this.value /= 2;
 },
};

const obj = {
 value: 1, //this type is an obj
 ...math,
};

obj.double();
console.log(obj.value);
```

```
//generics - depending on parameters intialization
type StateDescription<D, M> = {
 data: D;
 //this will get both D & M properties
 //methods use this, so binds ThisType here
 methods: M & ThisType<D & M>;
};

//initialize parameter, take out D and M, then return type as D&M
const createState = <D, M>(desc: StateDescription<D, M>): D & M => {
 return { ...desc.data, ...desc.methods };
```

```
};

const state = createState({
 data: { x: 0, y: 0 },
 methods: {
   moveBy(dx: number, dy: number) {
     this.x += dx;
     this.y += dy;
   },
 },
});
state.x = 10;
state.y = 20;
state.moveBy(5, 5); //15, 25
```

72. Await<T> unwrapped chained promise bk to its original type

```
//js await will wait for chained promise to resolve to final value
//js default behaviour to resolve chained promise

type WrappedPromise = Promise<Promise<Promise<string>>>;

const mainCustom = async () => {
 const single: Promise<string> = new Promise((res) => res("l4d135"));
 //i probably will never write like this sia
 // new Promise <initialize the promise return type>
 const triple: WrappedPromise = new Promise<Promise<Promise<string>>>((res) =>
   res(
     new Promise<Promise<string>>((res) =>
       res(
         new Promise<string>((res) => {
           res("Vin Diesel");
         }),
       ),
     ),
   ),
 );
```

```
 const singleResult = await single;
 console.log(singleResult); //14d135
 //the final type is just a string
 const tripleResult = await triple;
 console.log(tripleResult); // Vin Diesel
};

mainCustom();

//in a similar way, Await helps to wrap the chained promise
//awaitedResult is a string type
type AwaitedResult = Awaited<WrappedPromise>;
```

73. Type for uppercase/lowercase/capitalized

Ts has some built in utilities that directly built into compilers

```
type ABC = Uppercase<"abc">;
type abc = Lowercase<"ABC">;
type Abc = Capitalize<"abc">;
type cde = Uncapitalize<"cde">;
```

74. Mapped typed as a clause

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

/*
* {
 name: (value: string)=> void;
 age: (value: number)=> void
}
*/

type Setters = {
 [K in keyof State]: (value: State[K]) => void;
};

type SetProperty<K extends string> = `set${Capitalize<K>}`;
type ExampleName = SetProperty<"name">; //setName
```

```
type ExampleAge = SetProperty<"age">; //setAge
```

However oci ts does not work for below

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

/*
* {
 name: (value: string)=> void;
 age: (value: number)=> void
}
*/

type Setters = {
 //use as clause to convert key of an obj into a string type that wanted
 [K in keyof State as `set${Capitalize<K>}`]: (value: State[K]) => void;
};
```

More general getter and setter state
//some OCI ts does not work with below

```
type Setters<State> = {
 //use as clause to convert key of an obj into a string type that wanted
 // & string helps Captilize <> to recognize K as a strign
 //we only & need state key and the key is a string
 [K in keyof State & string as `set${Capitalize<K>}`]: (
   value: State[K],
 ) => void;
};

type Getters<State> = {
 [K in keyof State & string as `get${Capitalize<K>}`]: () => State[K];
};

/*
* {
 name: (value: string)=> void;
 age: (value: number)=> void
}
*/
```

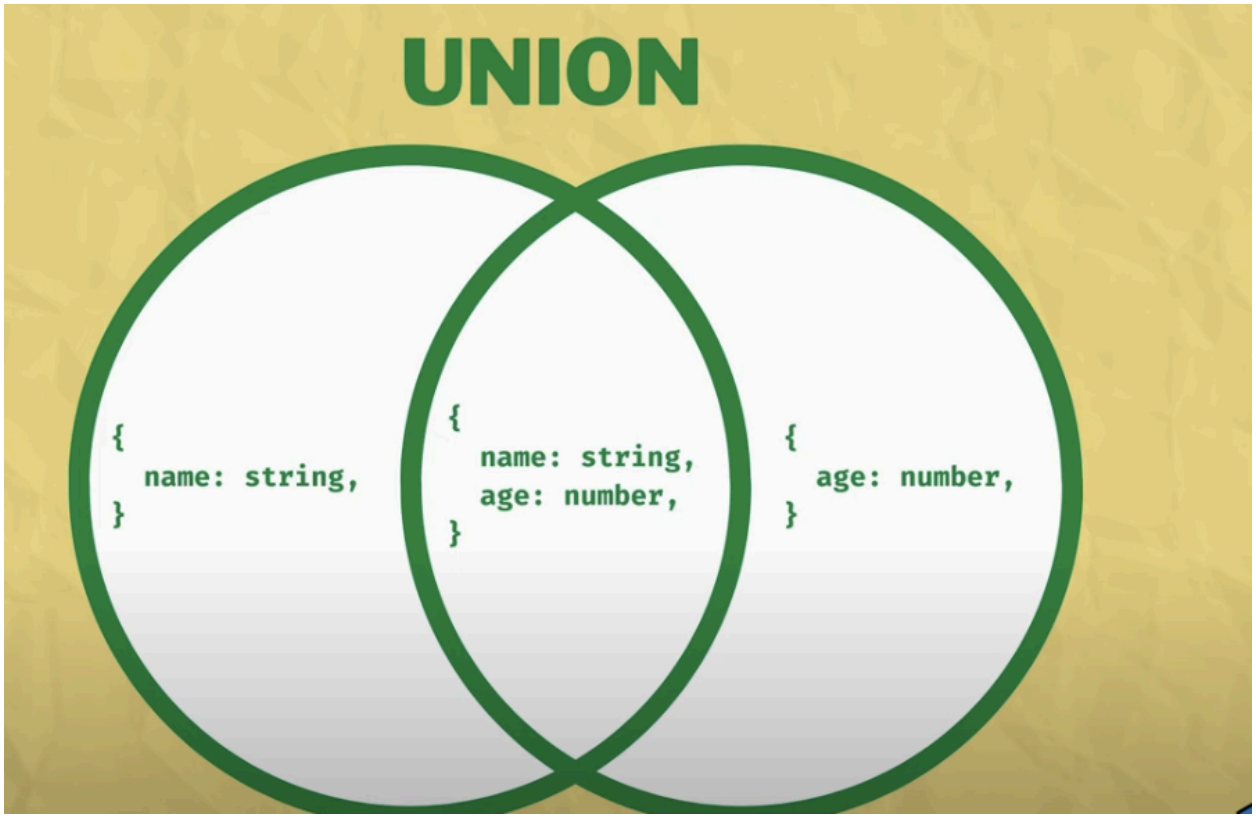
```ts
type SettersState = Setters<PersonState>;
//intersection of getter and setter
type Store<State> = Setters<State> & Getters<State>;
type PersonState = {
 name: string;
 age: number;
};

type PersonStore = Store<PersonState>;
const personStore: PersonStore;
personStore.setName("john");
personStore.setAge(22);
const nameCustomer: string = personStore.getName();
```

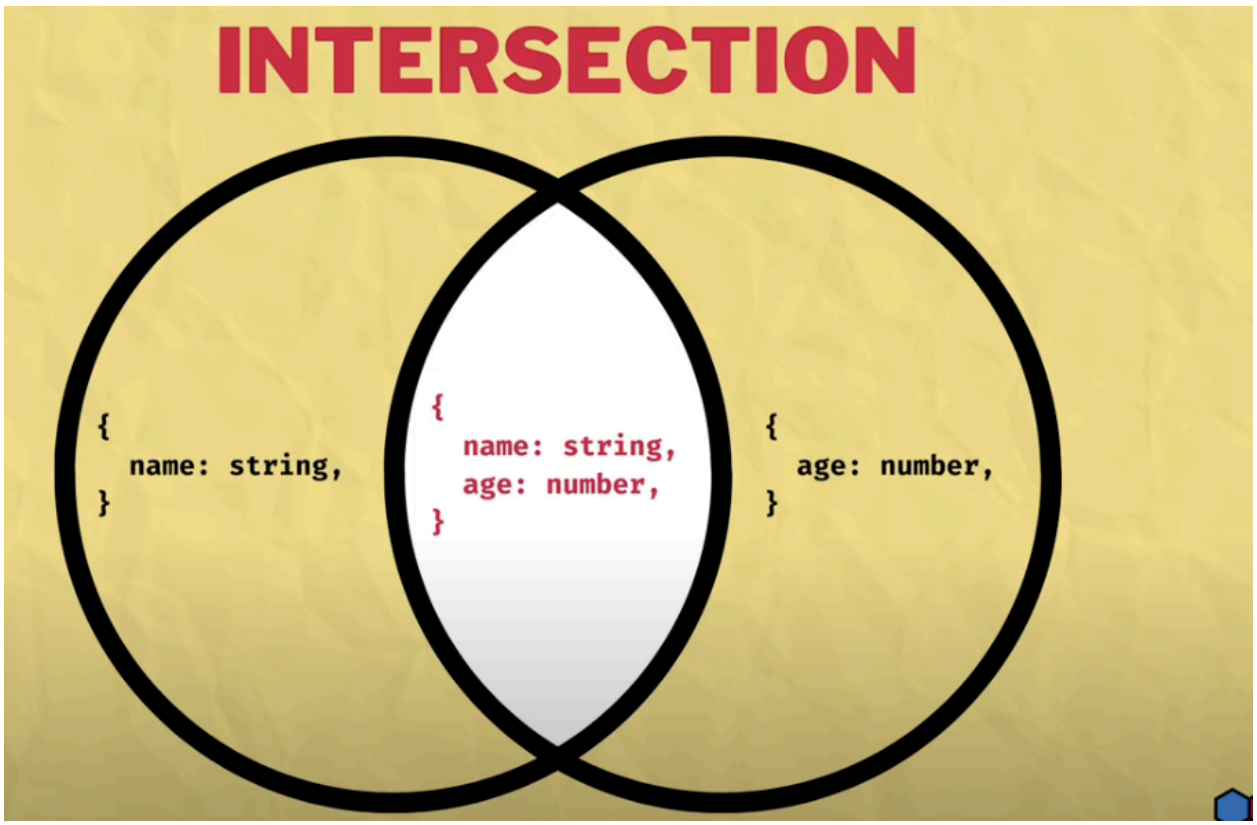
75. Ts union and intersection

Union and intersection - still not very clear

Union includes intersection while intersection is just intersection

- Below picture is not perfect
- We can assign something with name and age, but we cannot read them safely






```
type Name = { name: string };
```

```ts
type Age = { age: number };

type Union = Name | Age;
type Intersection = Name & Age;

const nameName = { name: "john" };
const age = { age: 132 };
const nameAndAge = { name: "jane", age: 22 };

let union: Union;
union = nameName;
union = age;
union = nameAndAge; //ts structural format

let intersection: Intersection;
intersection = nameAndAge;
intersection = nameName; //error
intersection = age; //error
```

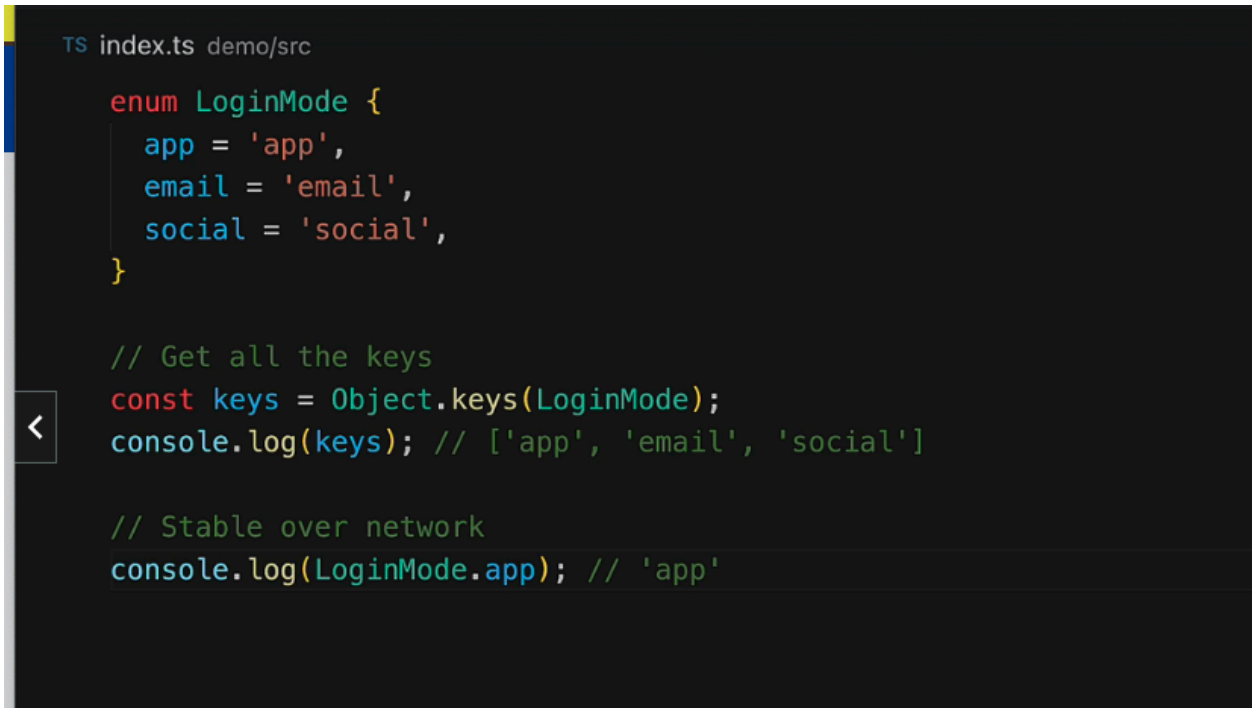
```ts
type Name = { name: string };
type Age = { age: number };

type Union = Name | Age;
type Intersection = Name & Age;

const nameName = { name: "john" };
const age = { age: 132 };
const nameAndAge = { name: "jane", age: 22 };

let union: Union;
union = nameName;
union = age;
union = nameAndAge; //ts structural format

const filter = (union: Union) => {
 if ("name" in union) {
   union.name;
 }

 if ("age" in union) {
   union.age;
```

```
 }

 if ("name" in union && "age" in union) {
   //ts support variable reassignment to union
   //but not for reading safely
   union.name; //error
   union.age; //error
 }
};
```

76. Ts enums are bad
Do not recommend for professional use

Auto numbering is prone to developer mistake
No type safety provided to enum (provide wrong number that ts will not complain)
Reverse look up - look up for both key and value, so object.keys might not return what u expected

What about string enums?
We r repeating our key in values (verbose)



Use type instead of enum

- Simple union of string literal

- Less typing
- Easy to integrate with function

You can extract keys from an object

77. Omit one keyv value pair

Omit 1 key value pair from the type

```
const extractFleetIdAfterCreateFleet = async (
        args: Omit<CreateFleetDetails, "inventoryLog"> & Partial<FleetInfo>,
): Promise<ResponseWithData<string>> => {
        let inventoryLog: CustomLog = null;
        let operationLog: CustomLog = null;
```

Omit<Type, Keys>

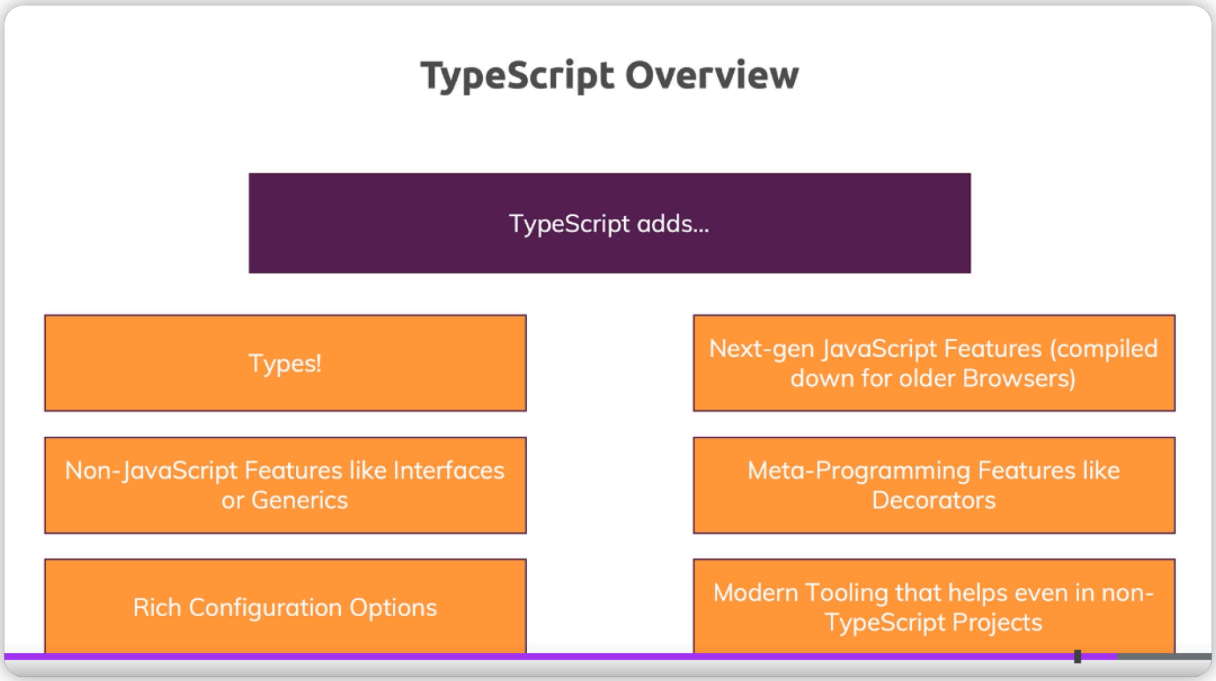
```
interface Todo {
 title: string;
 description: string;
 completed: boolean;
 createdAt: number;
}

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
 title: "Clean room",
 completed: false,
 createdAt: 1615544252770,
};

todo;
const todo: TodoPreview
```

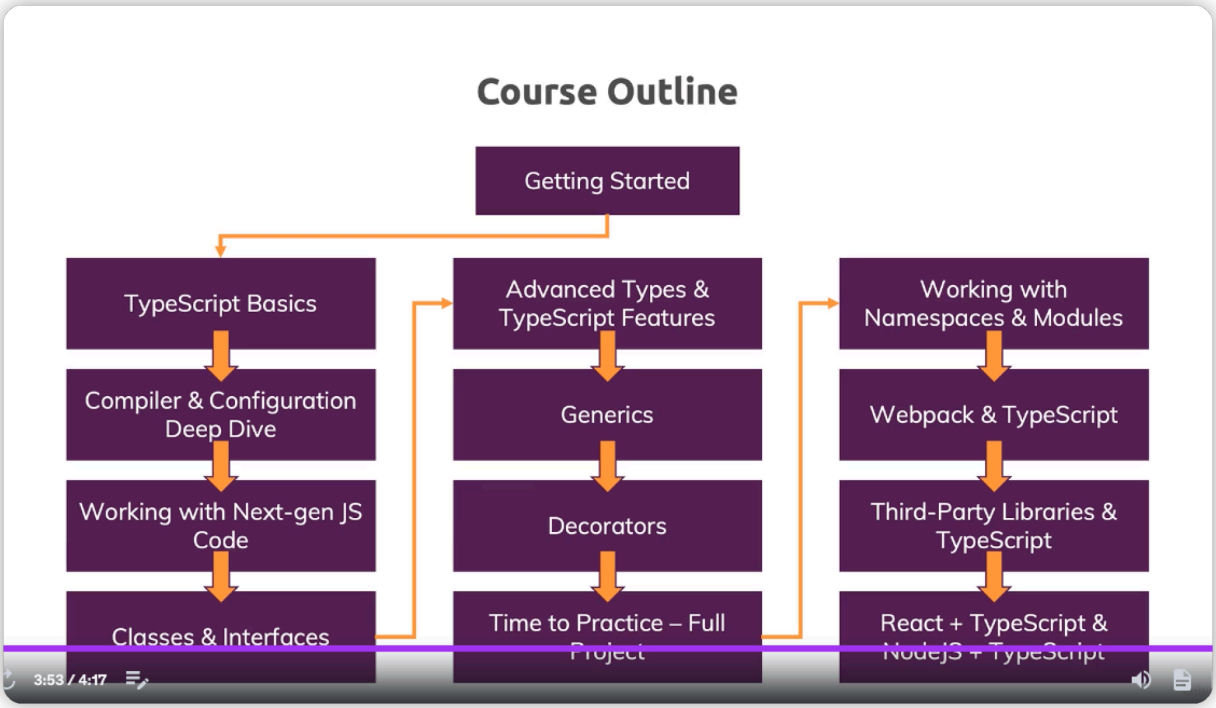
Introduction

Superset to JS
Compile ts into js
Add types - identify errors earlier before runtime and during development
Const input1 = document.getElementById(“num1”) ! as HTMLInputElement;

- ! Meant this will always yield an element, not null
- As HTMLInputElement as casting type for the html

After install typescript globally
Then we type tsc file.ts, will help us to convert from ts into js





Core type

- Number (no diff between integer and floats)
- String

- Boolean

App.ts

```
function add(n1: number, n2: number) {
   return n1 + n2;
}

//argument type string is not assignable to parameter number
const result = add("2", 3);
console.log(result);
```

TS only catch bugs in development, not run time

- JS use dynamic type, resolved at runt ime
- TS use static type, set during development

Boolean and string type

```
//added boolean and string type
function add(n1: number, n2: number, showResult: boolean, phrase: string) {
   if (showResult) {
       console.log(phrase + (n1 + n2))
   } else {
       return n1 + n2;
   }
}

const showResult = true;
const resultPhrase = "the result: ";
const result = add(2, 3, showResult, resultPhrase);
```

Type inference (ts can infer from its initialization, so no need to declare type for best practice)

```

//type inference
// below type is number3: number
let number3 = 3;

//below type is number4:4 (strict const)
```

```
const number4 = 4;

//not a good practice to declare below, as ts already know it is a number
let number5: number = 5;
//so write let number5 = 5 instead
```

Object - better let inference determine the type

```
//below is redundant to declare its type
let personA: {
   name: string;
   age: number
} = {
   name: "tom",
   age: 33
}


//ts infer below as an object person:{name: string; age: number}
//so is is redundant to declare it
let person = {
   name: "jerry",
   age: 23,
}

console.log(person.name)
```

Of course object types can also be created for **nested objects**.

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

```
let person = {
   name: "jerry",
   age: 23,
   // ts infer below as hobbies: string[];
   hobbies: ["run", "read"]
}

console.log(person.name)

//hobby: string is infered below
for (const hobby of person.hobbies) {
   //so directly ts allow us to access string related methods
   console.log(hobby.toUpperCase());
   //ts is yelling at us below because map is used for number, not string
   // console.log(hobby.map())
}
```

Tuple (fixed length array and fixed types) / array in Java

Tuple can catch all errors except for push

```
let person: {
   name: string;
   age: number;
   hobbies: string[];
   // tuple format
   role: [number, string]
```

```
} = {
   name: "jerry",
   age: 23,
   hobbies: ["run", "read"],
   //we want role only has length of 2 with fixed types, first as number, second as
string
   //then we would use tuple here
   role: [2, "admin"]
}

//tuple can catch all errors except for push
person.role.push("clerk");
//number cannot assign to string due to tuple type
person.role[1] = 3;
//tuple only allow 2 elements
person.role = [1, "admin", "boss"]
```

Enum - human readable identifier

```
// const ADMIN = 0;
// const CLERK = 1;

//enum could rep above as human readable identifier
//ADMIN=0, CLERK=1;
enum Role { ADMIN, CLERK };

//you can also assign your own index to the enum, either number or string
// enum Role { ADMIN=101, CLERK="Clerk" };

let person = {
   name: "jerry",
   age: 23,
   hobbies: ["run", "read"],
   // you can use role. to access different role easily
   role: Role.ADMIN
}

// you can use Role to access different roles easily
if (person.role === Role.ADMIN) {
   console.log(person.role)
```

```
}
```

Any type

Avoid to use it at any cost because it takes away ts functionality
Use it as last resort

Union type - a variable has 2 or more types defined

```
function combine(input1: number | string, input2: number | string) {
   let result;
   //ts does not know if you want to combine string or numbers, so be specific
   if (typeof input1 === "number" && typeof input2 === "number") {
       result = input1 + input2;
   } else {
       result = input1.toString() + input2.toString();
   }

   return result;
}

const combineAges = combine(3, 4);
console.log(combineAges);

const combinedNames = combine("tom", "alex");
console.log(combinedNames)
```

Literal type - literal text as the type

```
function combine(
   input1: number | string,
   input2: number | string,
   // literal type - where you type is very specific to literal words
   resultConversion: "as-number" | "as-text") {
   let result;
```

```
   if (typeof input1 === "number" && typeof input2 === "number" || resultConversion
=== "as-number") {
       result = +input1 + +input2;
   } else {
       result = input1.toString() + input2.toString();
   }
   return result;
}

const combineAges = combine(3, 4, "as-number");
console.log(combineAges);

const combineStringAges = combine("3", "4", "as-number");
console.log(combineStringAges);

const combinedNames = combine("tom", "alex", "as-text");
console.log(combinedNames)
```

Type alias - use variable to contain types and reuse in the codes

```
//use type alias

type Combinable = number | string;
type ConversionDecsriptor = "as-number" | "as-text";

function combine(
   input1: Combinable,
   input2: Combinable,
   resultConversion: ConversionDecsriptor) {
   let result;

   if (typeof input1 === "number" && typeof input2 === "number" || resultConversion
=== "as-number") {
       result = +input1 + +input2;
   } else {
       result = input1.toString() + input2.toString();
   }
   return result;
}
```

```
const combineAges = combine(3, 4, "as-number");
console.log(combineAges);

const combineStringAges = combine("3", "4", "as-number");
console.log(combineStringAges);

const combinedNames = combine("tom", "alex", "as-text");
console.log(combinedNames)
```

Type Aliases & Object Types

Type aliases can be used to "create" your own types. You're not limited to storing union types though - you can also provide an alias to a (possibly complex) object type.

For example:

```
type User = { name: string; age: number };
const u1: User = { name: 'Max', age: 30 }; // this works!
```

This allows you to avoid unnecessary repetition and manage types centrally.

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;
```

```
}
```

Function return type | void

```
//return type is infered as number
function add(n1: number, n2: number) {
   return n1 + n2;
}

//return is void, if there is no return key word
function printResult(num: number): void {
   console.log("result is " + num)
}

//return nothing is undefined
// function printResult(num: number): undefined {
//     console.log("result is " + num)
//     return;
// }

printResult(add(2, 3))

// let variable: undefined;
```

Function type - you can specifically describe the shape of your function

```
//you can describe the shape of your function, usually we describe the shape of fn to
pass in as parameter in another fn
type combineValues = (a: number, b: number) => number;

const useFunction = (a: number, b: number, fn: combineValues) => {
 return fn(a, b);
};
```

Callback function type

- You can define you call back function as parameter its shape
- When you set to void, meaning you are not using it in the current function
- When when you started using it, you can still return something in the end

```
//call back function


//define your call back function type
const addAndCall = (a:number, b:number, fn:(a:number)=>void) => {
 const result = a + b;
 fn(result);
}

addAndCall(1, 2, (result) => console.log(result));
```

Unknown type - better than any

```
//unknown type, you cannnot assign unknown type to a fixed type, they are totally
different
type variable = unknown;
type text = string;
let someVariable: variable = 3;
someVariable = "string";

if (typeof someVariable === "string") {
 const realText: text = someVariable;
}
```

Never function return type

```
//never return function, which 1)never reach to end point, or 2) always throw an error
const throwError = (error: string): never => {
 throw new Error(error); //when throwing an error, we never reach a end point
 // console.log(error); //this is not a never function
};

const keepProcessing = (): never => {
```

```
while (true) {
  console.log("keep printing");
}
};
```

Pro tip - it is all about the shape

Type fn: (number)=> number;

Shape of a function is only defined by its parameter and return results;

Typescript compiler and its configuration

Watch mode to detect auto changes and compile

Instead of typing “tsc app.ts”
Type “tsc app.ts - - watch”, and type control + c to quite this mode

Compile the entire file

- Type “tsc - -init” to generate a tsconfig.json file, and this tells ts to manage all ts files to where this json file lies
- Then we just type “tsc”, it will then convert all ts files in the folder into js files
- To enter into watch mode for the entire file, just type “tsc --watch”

Include/exclude certain files for compilation

```
{
 "compilerOptions": {
   ….
 },
 "exclude":[
   "node_modules" //default no need to write this
 ],
 "include":[
   "app.ts", "analytics.ts"
 ]
```

```
 // "files":[]//files are more specific, we only use this in very small files
}
```

Compiler option - target

Meaning that we compiling ts code into Es5 js syntax

```
{
 "compilerOptions": {

   "target": "ES5",
```

Compilation option - lib

Lib is commented out, set as default value (which has dom api, and follow es5)

```
  "target": "ES5",                                  /* Set the JavaScript language
version for emitted JavaScript and include compatible library declarations. */
   // "lib": [],
```

If you want to specify the lib, you need to uncoment it and add in libaries
Below are the default libraries you can set

```
"target": "ES5",                                  /* Set the JavaScript language
version for emitted JavaScript and include compatible library declarations. */
   "lib": [
     "dom",
     "es5",
     "DOM.Iterable",
     "ScriptHost"
   ],
```

Allow js and check js

Allow Js file to be compiled and checked for errors, if you do not want to use ts files

```
// "allowJs": true,                                  /* Allow JavaScript files to be a
part of your program. Use the 'checkJS' option to get errors from these files. */
   // "checkJs": true,
```

Source map - allow ts files to appear in source on browser

```
// "sourceMap": true,                                /* Create source map files for
emitted JavaScript files. */
```

Rootdir, outdir and others

- rootDir - specify src folder for tsx codes
- outDir - specify js folder after being compiled
- removeComments - remove/minify js file after compilation
- noEmit - do not compile all ts when you just want to check ts codes

```
// "rootDir": "./",                                  /* Specify the root folder within
your source files. */

   // "outDir": "./",                                   /* Specify an output folder
for all emitted files. */
   // "removeComments": true,                           /* Disable emitting comments.
*/
   // "noEmit": true,                                   /* Disable emitting files from
a compilation. */
```

Do not emit js file if there is errors in ts code

//default is false below, but you can set it to be true

```
    "noEmitOnError": false,
```

Turn on and off strict modes

- Strict: true turns on all type checking options below
-

```
/* Type Checking */
   "strict": true,                                /* Enable all strict
type-checking options. */
   // "noImplicitAny": true,                      /* Enable error reporting for
expressions and declarations with an implied 'any' type. */
   // "strictNullChecks": true,                   /* When type checking, take
into account 'null' and 'undefined'. */
   // "strictFunctionTypes": true,                /* When assigning functions,
check to ensure parameters and the return values are subtype-compatible. */
   // "strictBindCallApply": true,                /* Check that the arguments
for 'bind', 'call', and 'apply' methods match the original function. */
   // "strictPropertyInitialization": true,       /* Check for class properties
that are declared but not set in the constructor. */
   // "noImplicitThis": true,                     /* Enable error reporting when
'this' is given the type 'any'. */
   // "useUnknownInCatchVariables": true,         /* Default catch clause
variables as 'unknown' instead of 'any'. */
   // "alwaysStrict": true,                       /* Ensure 'use strict' is
always emitted. */
   // "noUnusedLocals": true,                     /* Enable error reporting when
local variables aren't read. */
   // "noUnusedParameters": true,                 /* Raise an error when a
function parameter isn't read. */
   // "exactOptionalPropertyTypes": true,         /* Interpret optional property
types as written, rather than adding 'undefined'. */
   // "noImplicitReturns": true,                  /* Enable error reporting for
codepaths that do not explicitly return in a function. */
   // "noFallthroughCasesInSwitch": true,         /* Enable error reporting for
fallthrough cases in switch statements. */
   // "noUncheckedIndexedAccess": true,           /* Add 'undefined' to a type
when accessed using an index. */
   // "noImplicitOverride": true,                 /* Ensure overriding members
in derived classes are marked with an override modifier. */
   // "noPropertyAccessFromIndexSignature": true, /* Enforces using indexed
accessors for keys declared using an indexed type. */
   // "allowUnusedLabels": true,                  /* Disable error reporting for
unused labels. */
   // "allowUnreachableCode": true,               /* Disable error reporting for
unreachable code. */
```

```
```

- NoImplicitAny - check if parameters has a declared type and report errors

```
-
-   //data is implicitly has any type
-   function send(data) {
-      console.log(data);
-   }
-
-   send("some data");
```

Strict null checking

Sometimes your app might fail, because you are doing operation on null. So below set to true will check if your codes have null.

```
"strictNullChecks": true,                          /* When type checking, take into
account 'null' and 'undefined'. */
```

! tell ts that as developer, we confirm it is not a null object, to avoid null checking error

```
const button = document.querySelector("button")!;

button.addEventListener("click", () => {
   console.log("clicked")
})
```

We can also use an if statement to check if it is null and ts understand will not complain

```
const button = document.querySelector("button");

if (button) {
   button.addEventListener("click", () => {
       console.log("clicked")
   })
}
```

Both ways work, but ! is a shorthand.

StrictBindCallApply, Always Strict

```
// "strictBindCallApply": true,                        /* Check that the arguments for
'bind', 'call', and 'apply' methods match the original function. */
   // "strictPropertyInitialization": true,             /* Check for class properties
that are declared but not set in the constructor. */
   // "noImplicitThis": true,                           /* Enable error reporting when
'this' is given the type 'any'. */
   // "alwaysStrict": true,                             /* Ensure 'use strict' is
always emitted. */
```

- Always strict - when we emit js files from ts, we always “use strict” in the js file
- StrictBindCallApply - when we use bind method, strict mode apply to it

```
const button = document.querySelector("button")!;

function callMessage(message: string) {
   console.log(message)
}

//if you use bind, only provide this, not a string message, ts will complain
if (button) {
   button.addEventListener("click", callMessage.bind(this, "a message"))
}
```

Code quality checks option

- noUnUsedLocals - cannot have unused variable in a local scope e.g. function
- unUsedPara - cannot have a para that is not used in a function
- noImplicitReturns - must have return for all possible paths

```
"noUnusedLocals": true,                            /* Enable error reporting when local
variables aren't read. */
   "noUnusedParameters": true,                     /* Raise an error when a function
parameter isn't read. */
   // "exactOptionalPropertyTypes": true,          /* Interpret optional property
types as written, rather than adding 'undefined'. */
   "noImplicitReturns": true,                      /* Enable error reporting for
codepaths that do not explicitly return in a function. */
```

See examples below

```
const button = document.querySelector("button")!;

function sum(n1: number, n2: number) {
   if (n1 + n2 > 0) {
       return n1 + n2;
   }
   //no implicit return for number <0
}

function callMessage(message: string, useUsedParameter: number) {
   // declared but never used
   let unUsedLocals = "ABC";
   console.log(message)
}

//if you use bind, only provide this, not a string message, ts will complain
if (button) {
   button.addEventListener("click", callMessage.bind(this, "a message", 23))
}
```

Modern js and ts -

let, const, arrow, destructing etc

```
// var and let
//var - local scope (not restricted), functional scope (restrict)
//let - local scope (restricted), functional scope (restricted)

//const - variable that never changes

function add(a: number, b: number) {
   var variable;
   variable = a + b;
   return variable;
}
```

```
if (true) {
   let localVariable = 3;
}
//able to access var, but not let
console.log(localVariable);
```

Different way of writing arrow f in ts

```
//different way of writing arrow function
const callMessage = (m: string) => console.log(m); //only define the parameter
//define the shape of the function right behind the function
const callMessage2: (m: string) => void = (message) => console.log(message);
```

Spread operator for array and object

Spread the elements and take out the brackets

```
//spread operator to spread array or object
const hobbies = ["walk", "sing"];
const others = ["shopping"]

const totalHobbies = [...hobbies, ...others];

//for object
const person = {
   name: "Alex",
   age: 31
}

const newPerson = { ...person }
```

Rest parameter for function when you have unlimited no of parameters

```
// rest parameter for function - for taking in unlimited parameters
const addNum = (...numbers: number[]){
```

```
    return numbers.reduce((ac, cv) => {
        return ac + cv
    }, 0)
}

//you can also limit no of parameters with tuple
const addNum2 = (...numbers: [number, number, number]){
    return numbers.reduce((ac, cv) => {
        return ac + cv
    }, 0)
}

let totalNum = addNum(1, 2, 3, 4);
console.log(totalNum);
```

```
//rest parameter to take in as many args as possibel and convert it into an array
//so for ...args is an array of number
const addMany = (...args: number[]) => {
 return args[0];
}
```

Array and object destructing - take out individual elements out of array and obj structure

```
//array and object destructing - break the structure of them and use individual
element
const hobbies = ["walk", "sing", "boxing"];

//take out h1=hobbies[0], h2=hobbies[1], otherH=[...other Elements]
const [h1, h2, ...otherH] = hobbies;
console.log(h1, h2, otherH);

const person = {
    firstName: "Alex",
    age: 31
}

//take out values by keys, you can also use : to use new alias
const { firstName: userName, age } = person;
console.log(userName, age, person)
//array and object destructuring
```

```
const hobbies = ["walk", "sing", "dance"];
const [h1, h2, ...rest] = hobbies;
console.log(rest); //["dance"];

const objectPerson = {
 name: "tom",
 age: 23,
};

const { name, age } = objectPerson; //use curly bracket to destructuing an object
console.log(name, age);
```

Class and Interface

Create a class in ts

```
class Department {
   name: string;
   constructor(n: string) {
       this.name = n;
   }
}

const accounting = new Department("Accounting")
console.log(accounting);
```

“this” key word - refer to instance created, or whoever is calling the method

```
class Department {
   name: string;

   constructor(n: string) {
       this.name = n;
   }

   //add this into parameter to describe this
   describe(this: Department) {
```

```
        console.log("department: " + this.name)
    }
}

const accounting = new Department("Accounting")
accounting.describe();

//this key word generally refer to the instance
//this key word refer to who is using the method

//copy the function to accountingCopy function
const accountingCopy = { name: "DUMMY", describe: accounting.describe }
//this.name will be nothing under acccountingCopy

accountingCopy.describe();
```

Access modifier - public and private

```
class Department {
    //public is default, no need to write the key word
    name: string;
    //employees variable is variable only accessible within the class
    //cannot directly mutate the variable
    private employees: string[] = [];

    constructor(n: string) {
        this.name = n;
    }

    //add this into parameter to describe this
    describe(this: Department) {
        console.log("department: " + this.name)
    }

    addEmployee(employee: string) {
        //validation logic
        this.employees.push(employee)
    }

    describeEmployee() {
        console.log(this.employees.length);
```

```
        console.log(this.employees)
    }
}

const accounting = new Department("Accounting")
accounting.describe();
accounting.addEmployee("Jerry")
accounting.addEmployee("tom")

//we do not want to access variable directly
//we should use getter and setter
//1. standard way of development for all engineers
//2. we can write validation logics within getter and setter
//below trigger an error because we have set the variable as private
//ts only check error during compilation, not during run time, as JS does not know
private key word
accounting.employees[2] = "Anna"
accounting.describeEmployee();
```

Shorthand initialization - init public type and variable within constructor method

```
class Department {
    // private id: string;
    // private name: string;
    private employees: string[] = [];

    //short hand initialization
    constructor(private id: string, public name: string) {
        // this.id = id;
        // this.name = name;
    }

    describe(this: Department) {
        console.log("department: " + this.id + " " + this.name)
    }

  …
}
```

```
const accounting = new Department("d1", "Accounting")
accounting.describe();
…
```

“Readonly” - can only read and cannot modify

```
class Department {
    // private readonly id: string;
    // private name: string;
    private employees: string[] = [];

    //readonly - can only read and cannot modify
    constructor(private readonly id: string, public name: string) {
        // this.id = id;
        // this.name = name;
    }

    …

    addEmployee(employee: string) {
        //cannot modify id because it is read only
        // this.id = "d2";
        this.employees.push(employee)
    }

    …
}

…
```

Inheritance - super is to call parent class constructor

```
class Department {
    // private readonly id: string;
    // private name: string;
    private employees: string[] = [];

    //readonly - can only read and cannot modify
```

```
    constructor(private readonly id: string, public name: string) {
        // this.id = id;
        // this.name = name;
    }

    describe(this: Department) {
        console.log("department: " + this.id + " " + this.name)
    }

    addEmployee(employee: string) {
        //cannot modify id because it is read only
        // this.id = "d2";
        this.employees.push(employee)
    }

    describeEmployee() {
        console.log(this.employees.length);
        console.log(this.employees)
    }
}

class ITDepartment extends Department {
    admins: string[];
    //constructor merge parent constructor para and child class para
    constructor(id: string, admins: string[]) {
        //super calls parent constructor method
        super(id, "IT");
        this.admins = admins;
    }
}


const it = new ITDepartment("d1", ["helpDesk", "Data"])
it.describe();
it.addEmployee("Jerry")
it.addEmployee("tom")

it.describeEmployee();

console.log(it)

class AccountingDepartment extends Department {
```

```
    //short hand initialization
    constructor(id: string, private reports: string[]) {
        super(id, "Accounting");
    }

    addReport(reportName: string) {
        this.reports.push(reportName)
    }

    printReports() {
        console.log(this.reports)
    }
}

const accounting = new AccountingDepartment("d2", []);
accounting.addReport("salary wrong...")
accounting.addReport("bonus wrong...")
accounting.printReports();
```

Override methods (from child class override), and protected key word

Private - only accessible within the class, but can be accessed using getter and setter
Protected - only accessible within the class and its children class

```
class Department {
    //change from "private" to "protect" below so that its children class can access
    protected employees: string[] = [];
    constructor(private readonly id: string, public name: string) {
    }
…
}


class AccountingDepartment extends Department {
    //short hand initialization
    constructor(id: string, private reports: string[]) {
        super(id, "Accounting");
    }
```

```
        //override addEmployees in the parent class
    addEmployee(name: string) {
        this.employees.push(name)
    }

  …
}

const accounting = new AccountingDepartment("d2", []);
…
accounting.addEmployee("Manu");
accounting.describeEmployee();
```

Getter and setter - diff from Java, get methodName, when calling do not use ()

```
class AccountingDepartment extends Department {
    private lastReport: string;

    //getter get and method name
    get mostRecentReport() {
        if (this.lastReport) {
            return this.lastReport
        }
        throw new Error("NO REPORTS FOUND")
    }

    set mostRecentReport(value: string) {
        if (!value) {
            return
        }
        this.addReport(value);
    }

    constructor(id: string, private reports: string[]) {
        super(id, "Accounting");
        this.lastReport = reports[0];
    }
```

```
    …

    addReport(reportName: string) {
        this.reports.push(reportName)
        this.lastReport = reportName;
    }

    …
}

const accounting = new AccountingDepartment("d2", []);

//calling setter function without ()
accounting.mostRecentReport = "salary wrong...";
accounting.addReport("bonus wrong...")
//calling getter function without ()
console.log(accounting.mostRecentReport)
…
```

Static method and property belong to class, detached from instances, typically used to group instances

```
class Department {
    //static property
    static fiscalYear = 2022;

    protected employees: string[] = [];
    constructor(private readonly id: string, public name: string) {
        //you can only access by className
        // console.log(Department.fiscalYear);
    }

    //static method
    static createEmployee(name: string) {
        return { name: name }
    }

  …
```

```
}


//calling static methods
const employee1 = Department.createEmployee("Max Manu");
console.log(employee1, Department.fiscalYear)
```

Abstract class - just a framework, not clear implementation

```
//abstract class, no instance instantiate
abstract class Department {
   //static property
   static fiscalYear = 2022;

   protected employees: string[] = [];
   constructor(protected readonly id: string, public name: string) {
   }

  …

   //abstract method, no implemention
   abstract describe(this: Department): void;

  …
}

class ITDepartment extends Department {
   admins: string[];
   constructor(id: string, admins: string[]) {
       super(id, "IT");
       this.admins = admins;
   }

   //having implementation details in child class, based on the abstract method
   describe() {
       console.log("Department ID: " + this.id)
   }
}

…
it.describe();
```

```
…

class AccountingDepartment extends Department {
   private lastReport: string;

   …

   //describe override method in parent class
   describe() {
       console.log("Department ID: " + this.id);
   }

   …
}

const accounting = new AccountingDepartment("d2", []);
console.log(accounting.describe())
```

Singleton (only 1 instance from 1 class), by using private constructor

- Private variable - only can access within class {} - for internal access only, we can use getter and setter for this
- Static - only can access usign class.static

```
class AccountingDepartment extends Department {
   private lastReport: string;
   //private (Only within class), static (access only through class)
   private static instance: AccountingDepartment;

   …
   //private constructor for singleton, instance cannot access it, only class can
access
//you can only call this constructor you when you in this class {} code
//so you cannot create new instance = new AccountingDepartment(__call constructor__)
   private constructor(id: string, private reports: string[]) {
       super(id, "Accounting");
       this.lastReport = reports[0];
```

```
    }

    //static allows us to access class based method and properties
    static getInstance() {
//this here means the class AccountingDepartment
        if (AccountingDepartment.instance) {
            return this.instance
        }
//access constructor privately within class
        this.instance = new AccountingDepartment("D2", [])
        return this.instance;
    }


…
}

//use class to access class level method, then construct privately within class
const accounting = AccountingDepartment.getInstance();
console.log(accounting);
```

Interface - structure for an object

Interface is usually used for an object

```
//1. interface to describe the structure of an object
interface Person {
 name: string;
 age: number;
 greet(phrase: string): void; //function is an object function greet() without func
word
}

const personObj: Person = {
 name: "tom",
 age: 23,
 greet(a) {
   console.log(a + " world");
 },
};
console.log(personObj.greet("hello"));
```

Use interface to constraint the shape of a class

```
interface Person {
 name: string;
 age: number;
 greet(phrase: string): void; //function is an object function greet() without func
word
}
//2. use interface to constrait class
class PersonClass implements Person {
 name: string;
 age: number;
 constructor(n: string, age: number) {
   this.name = n;
   this.age = age;
 }

 greet(phrase: string) {
   console.log(phrase);
 }
}
//we can personObj2 created can be interface structure
const personObj2: Person = new PersonClass("tom", 23);
console.log(personObj2.greet("haha"));
```

Why use interface?

- Person1 is defined as an interface, that it must has name and greet method
- We do not care if the Person class has other methods or variables, more than what interface constraints
- Sort of allowing us to describe the shape of our variable in a different way

User readonly on interface, and it affects the class as well

```
interface Greetable {
   //cannot set access modifier but readOnly
   readonly name: string;
   greet(phrase: string): void;
}


class Person implements Greetable {
   name: string;
   constructor(n: string) {
       this.name = n;
   }
   greet(phrase: string): void {
       console.log(phrase + this.name)
   }
}


let person1: Greetable;
//name is readOnly restricted by interface
person1.name = "Manu";
…
```

YOu can extends interface, or implement 2 or more interfaces on class

```
interface Named {
   readonly name: string;
}
interface Greetable extends Named {
   greet(phrase: string): void;
}


class Person implements Greetable {
   name: string;
   constructor(n: string) {
       this.name = n;
   }
   greet(phrase: string): void {
       console.log(phrase + this.name)
```

```
    }
}
```

USe Interface on function, as function === object in js

Interface is like contract, that classes have to adhere to

```
//use interface on function
interface PlusFn{
 //muse use : instead of => if you are using interface to define a function
 (a: number, b: number): number;
}

// type PlusFn2 = (a: number, b: number) => number;

//define function outside the real function below is a cleaner way of doing it
//easier to read
const plus: PlusFn = (a, b) => {
 return a + b;
}
```

Use? To mark para optional in function, or properties optional in interface and class

```
//when define attributes of an object, you can use ? to mark it option
interface Fruits {
 name: string;
 weight?: number;
 flavour: string;
}

class FruitBasket implements Fruits {
 name: string;
 // weight?: number; // no need to implement here, optional
 flavour: string; // flavour muse be declare here to fulfill the interface
 constructor(name: string) {
   this.name = name;
 }
}
```

```
const basket = new FruitBasket("jack fruit");
```

Advanced types

It works like interface extends

Intersection types - the common type properties of 2 types

- Below actually it is not intersection, more like union
- Because ElevatedEmployees must have props of name, privilege and startDate
- The intersection actually merge props from 2 types together

```
type Admin = {
   name: string;
   privilege: string[]
}

type Employee = {
   name: string;
   startDate: Date
}

//intersection types - the type where 2 types intersect each other
type ElevatedEmployee = Admin & Employee;

const e1: ElevatedEmployee = {
   name: "Max",
   privilege: ["free medical"],
   startDate: new Date()
}

type Combinable = string | number;
type Numeric = number | boolean;
//universal is a number - intersection of above
type Universal = Combinable & Numeric;
```

More type guards

Variable type guard using typeof

```
//use type guards together with union type
```

```
type Combinable = string | number;

const combine = (a: Combinable, b: Combinable) => {
 if (typeof a === "string" || typeof b === "string") {
   return a.toString() + b.toString();
 }
 return a + b;
};

console.log(combine("a", 2));
console.log(combine(1, 2));
console.log("a" + "b");
```

Object type guard using key in an object

```
//type guard for object check if key in an object
type UnknownEmp = Employee | FinanceEmp;

const printBenefits = (ep: UnknownEmp) => {
 if ("flex" in ep) {
   console.log("flex", ep.flex);
 }
 if ("insurance" in ep) {
   console.log("insurance", ep.insurance);
 }
};
printBenefits({ name: "tom", flex: 234 })
```

Class type guard using instanceof

```

//use type guard for class

//typeguard - if an object is an instanceof
class Car {
 drive() {
   console.log("car is driving")
 }
}

class Truck{
```

```
 drive() {
   console.log("truck is driving")
 }
 loadCargo() {
   console.log("truck is loading cargo")
 }
}

const car = new Car();
const truck = new Truck();
//you can use class to define a type as well, just like how you define an object
type vehicle = Car | Truck;
const useVehicle = (v: vehicle)=>{
 v.drive();
 if (v instanceof Truck) {
   v.loadCargo();
 }
}
```

Discriminating unions - identify which type
Union type + type discrimination to find out which one

```
//union type + discriminating type to differentiate which one
interface Bird {
 type: "bird";
 flySpeed: number;
}

interface Horse {
 type: "horse";
 runningSpeed: number;
}

type Animal = Bird | Horse;
const animalSpeed = (animal: Animal) => {
 switch (animal.type) {
   case "bird":
     console.log(animal.flySpeed);
     break;
   case "horse":
```

```
    console.log(animal.runningSpeed);
 }
};

animalSpeed({ type: "horse", runningSpeed: 2233 });
```

Type casting - tell ts what dom element is it, as ts do not read html files

Type casting (Below does not work in react)

- You can use <> to put before the variable
- Or you can use “as “

```
//type casting on dom elements, as TS does not read our html files

//1. ts only know it is a htmlElement, does not know if it is an input, you need to
tell ts
//2. use "!" to tell ts it is definitely not a null
const htmlInputElement = <HTMLInputElement>document.getElementById("text-input")!;
htmlInputElement.value = "Hi there";

//alternative way
// const htmlInputElement2 = document.getElementById("text-input2")! as
HTMLInputElement;
// htmlInputElement2.value = "Hi there2";

const htmlInputElement2 = document.getElementById("text-input2");
if (htmlInputElement2) {
   (htmlInputElement2 as HTMLInputElement).value = "Hi there2";
}
```

```
const TsPractice = (): React.ReactElement => {
 //when using hooks, specify type after useHooks
 const inputElement = React.useRef<HTMLInputElement>(null);

 console.log(inputElement?.current?.value);

 return (
   <>
```

```
    <h1>tspractice</h1>
    <input ref={inputElement} />
   </>
 );
};
```

Do not know how many key value pair

```
//index properties - we do not know how many prop, and we do not know the exact prop name
//do not know how many key value pair
interface ErrorShape {
 [key: string]: string;
}

//Error bag can have multiple key value pair
const ErrorBag: ErrorShape = {
 name: "error",
 age: "error",
};
```

Function overload - changing signatures of the function to have different types of the same function

```
//function overload
//define pattern 1 and 2, then implementation including patttern 1 and 2
function addNumber(a: number, b: number): number;
function addNumber(a: number, b: number, c: number): number;
function addNumber(a: number, b: number, c?: number) {
 if (a !== undefined && b !== undefined && c !== undefined) {
   return a + b + c;
 } else {
   return a + b;
 }
}
//we can use both signatures.
```

```
addNumber(1, 2);
addNumber(1, 2, 3);
```

Optional chaining using ? to check if it is null before continuing

```
//optional chaining -? helps to prevent returning of null to cause an error, it will return undefined if nothing is found
const employee = {
 name: "Tom",
 age: 23,
 family: {
   sister: "joanne",
   houses: {
     address: "122",
     address2: "233"
   }
 }
}

console.log(employee.family?.houses?.address);
```

Nullish coalescing - check if undefine and null only

```
// The nullish coalescing operator(??) is a logical operator that returns its right - hand side operand when its left - hand side operand is null or undefined,
// and otherwise returns its left - hand side operand.
const foo = null ?? "DEFAULT";
console.log(foo); //return "DEFAULT";
const goo = 0 ?? "DEFAULT";
console.log(goo); // return 0;
```

Generics - flexible and reusable codes (exist in other languages)

https://replit.com/@ChenJerry3/what-is-promise#index.js

```
//what is promise (take time produce and consume code)
//Promise.state return state of the async operation (pending, fulfilled, reject);
//Promise.result return result of the ops (undefined, result value, error object);

const handleFulFillFn=(value)=>{
  console.log(value);
}

const handleRejectFn=(error)=>{
  console.log(error)
}

const myPromise = new Promise((resolve,reject)=>{
  //produce code
  let x = 0;

  if(x===0){
    resolve("success") //this produce Promise.result = "success";
  } else {
    reject("error"); //this produce an error object
  }
})

console.log(myPromise);//why returning undefined

//consume result
myPromise.then(handleFulFillFn, handleRejectFn);
```

Built in generic type

```
// const names: string[] = [];
const names: Array<string> = []
names[0].split(" ");

//<string> tell the promise return a string
const promise: Promise<string> = new Promise((resolve, reject)=>{
   setTimeout(()=>{
       resolve("Success")
   },2000)
})
```

```
//then to process the return data
promise.then((data)=>{
   data.split(" ");
})
```

Function generics - let parameter type inherit when you invoking the function

Object.assign merge together, if there is common target. It will merge together, taking new value from source
const target = { a: 1, b: 2};
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }

```
function merge0(objA: object, objB: object) {
   return Object.assign(objA, objB);
}

const mergeName0 = merge0({ name: "max" }, { age: 33 })
//ts does not know mergeName has a name prop, because we did not inform its type
console.log(mergeName0.name)


//use generics - T is kind of any type, depending on how you use the funciton,
//T type is diff from U Type, but in the end they have an intersection type
function merge1<T, U>(objA: T, objB: U) {
   return Object.assign(objA, objB);
}

//mergeName:{name: string}&{age: number} - intersection (thought it is the common
area)
const mergeName1 = merge1({ name: "max", hobbies: ["run"] }, { age: 33 })
//you can also fix the return type when you are using the function
```

```
const mergeName2 = merge1<{ name: string }, { age: number }>({ name: "max" }, { age:
33 })
//inform ts its type, then we can access the name
console.log(mergeName1.name)
```

```

//a better way is to use generics, when u started using it, it will inherit what you
have started

const merge =<T, U> (a: T, b: U) => {

 return Object.assign(a, b);

}



//1. ts infer the type

const mergedName = merge({ name: "tom" }, { age: 23 });

console.log(mergedName.name)



//2. you specify the usage type

const mergedName2 = merge<{name: string},{age: number}>({ name: "tom" }, { age: 23 });
```

Generic function constraint using extends key word

```
//just like class, U extends to object must have the prop of an object
function merge1<T extends object, U extends object>(objA: T, objB: U) {
   return Object.assign(objA, objB);
}

//so 33 is not a valid parameter of U
const mergeName1 = merge1({ name: "max", hobbies: ["run"] }, 33)
```

```
console.log(mergeName1.name)
```

```
//you can still constraints your generic type by using the extends key word
//below constraits t and u must be an object
interface startingType {[key:string]: unknown}
const mergeSome =<T extends startingType, U extends startingType>(a: T, b: U) => {
 return Object.assign(a, b);
}

// const mergeNames2 = mergeSome("a", "b");
```

Generic function para can be anything but must have a length prop

```
//an object that has length prop so that ts allows element.length
interface Lengthy {
   length: number
}



//T must have length prop
function countAndDescribe<T extends Lengthy>(element: T): [T, string] {
   let descriptionText = "Got no text";
   if (element.length === 1) {
       descriptionText = "got 1 text";
   } else { element.length > 1 } {
       descriptionText = "got " + element.length + " text";
   }

   return [element, descriptionText]
}

console.log(countAndDescribe(["hi", "there"]))
```

```
//must have a length property, array has length property
interface Lengthy {
 length: number;
}
const countAndDescribe = <T extends Lengthy>(element: T): [T, string] => {
 let descriptor = "got some value";
```

```
 if (element.length === 1) {
   descriptor = "got 1 value";
 } else if (element.length === 2) {
   descriptor = "got 2 value";
 }

 return [element, descriptor];
};
//anything that should have length property, string, array or object
console.log(countAndDescribe("123"));
console.log(countAndDescribe([1, 2]));
console.log(countAndDescribe({name: "tom", length: 2}))
```

Keyof - indicate one generic type is the key of another generic type

```
//keyof - indiciate 1 type is the keyof another generic type
function extractAndConvert<T extends object, U extends keyof T>(obj: T, key: U) {
   return obj[key];
}
//ts will complain name does not exist in object T
extractAndConvert({ age: 20 }, "name")
```

```
//use keyof to tell ts one type a key in another generics
const extract = <T extends startingType, U extends keyof T>(
 object: T,
 key: U,
) => {
 return object[key];
};
//the key must exist in the object
console.log(extract({ name: "Tom" }, "name"));
console.log(extract({ name: "Tom", age: "23" }, "age"));
```

Class generics

Generics allow us to have

- Flexibility (we also provide constraints)
- But also type safety

```
//restrict the generic 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;
       this.data.splice(this.data.indexOf(item), 1); //indexOf return -1 if did not
find
   }

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

const textStorage = new DataStorage<string>();
textStorage.addItem("Max")
textStorage.addItem("violet")
textStorage.removeItem("violet")
console.log(textStorage.getItems())

const numberStorage = new DataStorage<number>();

//do not use above class for obj
// const objStorage = new DataStorage<object>();
// let referObj = { name: "max" };
// objStorage.addItem(referObj);
// objStorage.addItem({ name: "manu" });
// //cannot remove {name: "max"} because obj is an reference
// objStorage.removeItem(referObj);
// console.log(objStorage.getItems())
```

```
//extends helps to constraint T a bit

class DataStorage<T extends string | number | boolean> {

 //using T to define our array storage

 private data: T[] = [];



 addItem(item: T) {

   this.data.push(item);

 }

 removeItem(item: T) {

   if (this.data.indexOf(item) === -1) return;

   this.data.splice(this.data.indexOf(item), 1);

 }

 getItems() {

   return [...this.data];

 }

}



const storage = new DataStorage<string>();

// storage.addItem(3); when defined class as a stirng, then cannot take number

storage.addItem("we");
```

```
storage.getItems();
```

Utility types with generics - partial type at beginning, and ReadOnly for locking the variable for edits

```
//built in utility partial type
interface CourseGoal {
   title: string;
   description: string;
}

function createCourse(title: string, description: string): CourseGoal {
   // return { title: title, description: description }
   //use generics with partial keyword
   let course: Partial<CourseGoal> = {};
   course.title = title;
   course.description = description;
   return course as CourseGoal;
}
```

```
//partial type allows us to slowly build up the object we want
interface CourseDescription {
 title: string;
 description: string;
}

const createCourse = (
 title: string,
 description: string,
): CourseDescription => {
 //partial allows us to define course as an empty {}, start adding attribute to the
object
 const course: Partial<CourseDescription> = {};
 course.title = title;
 course.description = description;
 //partial<CourseDescription> is not the type CourseDescription, so use casting at the
end
 return course as CourseDescription;
```

```
};
```

Read only to lock variables

```
const names: Readonly<string[]> = ["tom", "jerry"];
//below cannot push because it is a read only array
names.push("max");
```

Generic locks in the type, while union can be a mixture of any types/above generics lock in the type

```
//union type is flexible to be any data type, and array can be a mixtuer of number,
string, boolean
class DataStorage2 {
   private data: (string | number | boolean)[] = [];
}

//generic type lock in the type when you create the instance, it has be either an
array of number, string or boolean
class DataStorage<T extends string | number | boolean>{
   private data: T[] = [];
```

Decorator - help other developers when they are using your class

Decorators **provide a way to add both annotations and a meta-programming syntax for class declarations and members**. Decorators are a stage 2 proposal for JavaScript and are available as an experimental feature of TypeScript. NOTE Decorators are an experimental feature that may change in future releases.

In config file, we turn on this

```
"experimentalDecorators": true,                    /* Enable experimental support for
TC39 stage 2 draft decorators. */
```

First decorator

```
//decorator is useful for meta programming - writing codes easier for other developers
//it helps to guarantee our class is used correctly
function Logger(constructor: Function) {
   console.log("Logging...");
   console.log(constructor)
}

@Logger
class PersonClass {
   name = "Max";

   constructor() {
       console.log("creating an object...")
   }
}

const pers = new PersonClass(); //calling constructor
console.log(pers);

/*
decorator using class when it is defined not when it is instantiated

Logging...
app.js:10 class PersonClass {
   constructor() {
       this.name = "Max";
       console.log("creating an object...");
   }
}
app.js:15 creating an object...
app.js:22 PersonClass {name: 'Max'}
*/
```

Decorator factory to return a deco function - allow for customization

```
//decorator factory returns a deco function
function Logger(logString: string) {
   return function (constructor: Function) {
```

```
        console.log(logString);
        console.log(constructor)
    }
}


@Logger("LOGGING Person-")
class PersonClass {
    name = "Max";

    constructor() {
        console.log("creating an object...")
    }
}

const pers = new PersonClass(); //calling constructor
console.log(pers);
```

Userful decoration - make it template for others to import

- Below will output h1 tag on the browser

```
unction WithTemplate(template: string, hookId: string) {
    //use _ to tell deco that we do not care about the paramter
    return function (_: Function) {
        let hookEl = document.getElementById(hookId);
        //pass null checker
        if (hookEl) {
            hookEl.innerHTML = template;
        }
    }

}
// @Logger("LOGGING Person-")
@WithTemplate("<h1>With Template</h1>", "app")
class PersonClass {
    name = "Max";
}
```

- Decorator is an utility tool for other developers to use together with the class created
- It usually comes with 1) template 2) place to render, so that we could reuse this decoration again, without rewriting the html/template in this case

```
//decor comes with 1. template 2. place to render, similar to angular
//decorator is an utility tool we can provide to other developer for them to use with
the class we created
function WithTemplate(template: string, hookId: string) {
   //use _ to tell deco that we do not care about the paramter
   return function (constructor: any) {
       let hookEl = document.getElementById(hookId);
       //pass null checker
       let p = new constructor();
       if (hookEl) {
           hookEl.innerHTML = template;
           //render <h1>Max</h1> on the browser
           document.querySelector("h1")!.textContent = p.name;
       }
   }

}
// @Logger("LOGGING Person-")
@WithTemplate("<h1>With Template</h1>", "app")
class PersonClass {
   name = "Max";
…
}
```

Multiple decorator - run factory top to bottom, run deco f bottom up

```
function Logger(logString: string) {
   console.log("Logger factory")
   return function (constructor: Function) {
       console.log(logString);
       console.log(constructor)
   }
}



function WithTemplate(template: string, hookId: string) {
```

```
    console.log("Template factory")
    return function (constructor: any) {
        console.log("creating template...")
        let hookEl = document.getElementById(hookId);
        let p = new constructor();
        if (hookEl) {
            hookEl.innerHTML = template;
            document.querySelector("h1")!.textContent = p.name;
        }
    }

}
//run factory, logger, then withTemplate
//run deco f, withTemplate then Logger
@Logger("Logging...")
@WithTemplate("<h1>With Template</h1>", "app")
class PersonClass {
    name = "Max";
```

Use decorator on property

```
function Log(target: any, propertyName: string) {
    console.log("Property decorator")
    //target is the constructor, propertyName is "title"
    console.log(target, propertyName)

}

class Product {
    @Log
    title: string;
    private _price: number;
    constructor(t: string, p: number) {
        this.title = t;
        this._price = p;
    }

    set price(val: number) {
        if (val > 0) {
            this._price = val;
```

```
        } else {
            throw new Error("val for price must be postive!")
        }
    }

    getFinalPrice(taxRate: number) {
        return this.price * (1 * taxRate)
    }
}

// Property decorator
// app.js:57
// {constructor: ƒ, getFinalPrice: ƒ}
// constructor: class Product
// getFinalPrice: ƒ getFinalPrice(taxRate)
// set price: ƒ price(val)
// [[Prototype]]: Object
```

Accessor, method decorator and parameter decorator

```
//decorator for accessor (protoType, name of the method, description of the property)
function Log2(target: any, name: string | symbol, descriptor: PropertyDescriptor) {
    console.log("accessor decorator")
    console.log(target);
    console.log(name);
    console.log(descriptor)
}

// accessor decorator
// app.js:62 price
// app.js:63
// {get: undefined, enumerable: false, configurable: true, set: ƒ}
// configurable: true
// enumerable: false
// get: undefined
// set: ƒ price(val)
// [[Prototype]]: Object

function Log3(target: any, name: string | symbol, descriptor: PropertyDescriptor) {
    console.log("method decorator")
    console.log(target);
```

```
    console.log(name);
    console.log(descriptor)
}

// method decorator
// app.js:77
// {constructor: ƒ, getFinalPrice: ƒ}
// constructor: class Product
// getFinalPrice: ƒ getFinalPrice(taxRate)
// set price: ƒ price(val)
// [[Prototype]]: Object
// app.js:78 getFinalPrice
// app.js:79
// {writable: true, enumerable: false, configurable: true, value: ƒ}
// configurable: true
// enumerable: false
// value: ƒ getFinalPrice(taxRate)
// writable: true
// [[Prototype]]: Object

//parameter decorator(target, name of the method, position of the parameter=0)
function Log4(target: any, name: string | symbol, position: number) {
    console.log("parameter decorator")
    console.log(target);
    console.log(name);
    console.log(position)
}

class Product {
    @Log
    title: string;
    private _price: number;
    constructor(t: string, p: number) {
        this.title = t;
        this._price = p;
    }

    @Log2
    set price(val: number) {
        if (val > 0) {
            this._price = val;
        } else {
```

```
            throw new Error("val for price must be postive!")
        }
    }
    @Log3
    getFinalPrice(@Log4 taxRate: number) {
        return this.price * (1 * taxRate)
    }
}
```

When did decorator execute

- Not when instance is instantiate, it is when the class is created, we do not need to instance to be created to do something

JS class is syntatic sugar

```
//f as constructor
function student(name, age){
  this.name = name;
  this.age = age;
}

const A = new student("jerry", 23);
console.log(A)
//student { name: 'jerry', age: 23 }

//a class is just a constructor in js
```

Classes are also a special type of functions using prototype-based inheritance. To declare the class we need to use the class keyword.

```
class Student{
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
}
//new Student is actually calling constructor function
const B = new Student("Tom","22")
console.log(B)
```

Returning a new class from the decorator of an old class

Still a bit blurred on below

```
function WithTemplate(template: string, hookId: string) {
   console.log("Template factory")
   //originalConstructor is T type
   //T type that accept any arg (because we are inserting new arg returned class) and
return an obj with "name" key
   return function <T extends { new(...args: any[]): { name: string }
}>(originalConstructor: T) {
       //return a new class extends f/constructor the same thing
       return class extends originalConstructor {
           //it is arg, but we do not use it , so write _ instead
           constructor(..._: any[]) {
               //calling original constructor with super, save original class
               super();
               //below extra logic when the class is instantiated
               console.log("creating template...")
               let hookEl = document.getElementById(hookId);
               if (hookEl) {
                   hookEl.innerHTML = template;
                   //this.name is to get instance of the class
                   document.querySelector("h1")!.textContent = this.name;
               }
           }
       }
   }

}
//run factory, logger, then withTemplate
//run deco f, withTemplate then Logger
@Logger("Logging...")
@WithTemplate("<h1>With Template</h1>", "app")
class PersonClass {
   name = "Max";

   constructor() {
       console.log("creating an object...")
   }
}

const pers = new PersonClass(); //calling constructor
```

```
console.log(pers);
```

Read more on descriptor below
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

For access and method decorator, you can return a new descriptor.

Use bind to tell js what is the meaning of “this”

```
class Printer {
   message = "this works";

   showMessage() {
       console.log(this.message);
   }
}

const p = new Printer();
const button = document.querySelector("button")!;
//below will not work, the "this" in the showMessage f refers to button in event
listener
// button.addEventListener("click", p.showMessage)

//now we use showMessage f.bind(p object) - meaning to use p object
button.addEventListener("click", p.showMessage.bind(p))
```

Autobind this to the method in a class, without bind(this)

```
function AutoBind(_: any, _2: string, descriptor: PropertyDescriptor) {
   const originalMethod = descriptor.value;
   const adjDescriptor: PropertyDescriptor = {
       configurable: true,
       enumerable: false,
       get() {
           //this here refers to the object calling get()
           const boundFn = originalMethod.bind(this);
           return boundFn;
       }
```

```
    }
    //will override old descriptor
    return adjDescriptor;
}

class Printer {
    message = "this works";
    //decorator
    @AutoBind
    showMessage() {
        console.log(this.message);
    }
}

const p = new Printer();
const button = document.querySelector("button")!;
//below will not work, the "this" in the showMessage f refers to button in event
listener
// button.addEventListener("click", p.showMessage)

//now we use showMessage f.bind(p object) - meaning to use p object
button.addEventListener("click", p.showMessage)
```

Did not finish session 116 - using decorator for validation

- Did not see the point of writing this myself
- There is 3rd party library already has the package
- I can just use below

https://www.npmjs.com/package/class-validator

More on decorator
https://www.typescriptlang.org/docs/handbook/decorators.html

Drag and drop project with ts

Index.html

- Template is avail in html, it does not show on browser, we use it by strip off the template bracket

- And then insert into “app” id

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <meta http-equiv="X-UA-Compatible" content="ie=edge" />
   <title>ProjectManager</title>
   <link rel="stylesheet" href="app.css" />
   <script src="dist/app.js" defer></script>
 </head>
 <body>
   <template id="project-input">
     <form>
       <div class="form-control">
         <label for="title">Title</label>
         <input type="text" id="title" />
       </div>
       <div class="form-control">
         <label for="description">Description</label>
         <textarea id="description" rows="3"></textarea>
       </div>
       <div class="form-control">
         <label for="people">People</label>
         <input type="number" id="people" step="1" min="0" max="10" />
       </div>
       <button type="submit">ADD PROJECT</button>
     </form>
   </template>

   <template id="single-project">
     <li></li>
   </template>

   <template id="project-list">
     <section class="projects">
       <header>
         <h2></h2>
       </header>
       <ul></ul>
     </section>
   </template>
```

```
    <div id="app"></div>
  </body>
</html>
```

Use a class to instantiate a browser element

```
class ProjectInput {
    //template with template brackets
    templateElement: HTMLTemplateElement;
    //where to put element
    hostElement: HTMLDivElement;
    //element with template brackets gone
    element: HTMLElement;
    titleInputElement: HTMLInputElement;
    peopleInputElement: HTMLInputElement;
    descriptionInputElement: HTMLInputElement;


    constructor() {
        this.templateElement = document.getElementById("project-input")! as
HTMLTemplateElement;
        this.hostElement = document.getElementById("app")! as HTMLDivElement;
        //import the template node
        const importedNode = document.importNode(this.templateElement.content, true);
        //strip off the template open and closing brackets
        this.element = importedNode.firstElementChild as HTMLFormElement;
        //call this attach when instantiated immediately
        this.element.id = "user-input";
        this.titleInputElement = this.element.querySelector("#title") as
HTMLInputElement;
        this.descriptionInputElement = this.element.querySelector("#description") as
HTMLInputElement;
        this.peopleInputElement = this.element.querySelector("#people") as
HTMLInputElement;
        //catch user input
        this.configure()

        //insert the html codes
        this.attach()
```

```
    }

    private gatherUserInput(): [string, string, number] | void {
        const enteredTitle = this.titleInputElement.value;
        const enteredDescription = this.descriptionInputElement.value;
        const enteredPeople = this.peopleInputElement.value;

        const titleValidatable: Validatable = { value: enteredTitle, required: true };
        const descriptionValidatable: Validatable = { value: enteredDescription,
required: true, minLength: 5 };
        const peopleValidatable: Validatable = { value: +enteredPeople, required: true,
min: 1, max: 5 }

        if (
            // enteredTitle.trim().length === 0 ||
            // enteredDescription.trim().length === 0 ||
            // enteredPeople.trim().length === 0
            !validate(titleValidatable) ||
            !validate(descriptionValidatable) ||
            !validate(peopleValidatable)) {
            alert("Invalid input");
            return;
        } else {
            return [enteredTitle, enteredDescription, +enteredPeople];
        }
    }

    private clearInputs() {
        this.titleInputElement.value = "";
        this.descriptionInputElement.value = "";
        this.peopleInputElement.value = "";
    }

    @AutoBind
    private submitHandler(e: Event) {
        e.preventDefault();
        const userInput = this.gatherUserInput();
        if (Array.isArray(userInput)) {
            const [title, desc, people] = userInput;
            console.log(title, desc, people);
            this.clearInputs();
        }
```

```
    }

    private configure() {
        //bind(this) to point at class/instance
        this.element.addEventListener("submit", this.submitHandler)
    }

    private attach() {
        //insert the form into the <div "app"> afterbegin
        this.hostElement.insertAdjacentElement("afterbegin", this.element)
    }
}

//when user load js script, it creates an instance, then on this instance/object,
users can interact with browser
//this instance will keep track of everything that user did
const projInput = new ProjectInput();
```

Reusable validation function

```
interface Validatable {
    value: string | number;
    //? makes it optional, possibly undefined
    required?: boolean;
    minLength?: number;
    maxLength?: number;
    min?: number;
    max?: number;
}

function validate(validateInput: Validatable) {
    let isValid = true;
    if (validateInput.required) {
        //toString to convert string/number to string to avoid ts error
        isValid = isValid && validateInput.value.toString().trim().length !== 0;
        // console.log("required")
    }
    //length checking is only for string && minLength is not null or undefined (avoid 0
input from user)
    if (validateInput.minLength != null && typeof validateInput.value === "string") {
        isValid = isValid && validateInput.value.length >= validateInput.minLength;
```

```
        // console.log("min 1")
    }
    if (validateInput.maxLength != null && typeof validateInput.value === "string") {
        isValid = isValid && validateInput.value.length <= validateInput.maxLength;
        // console.log("max 1")
    }
    if (validateInput.min != null && typeof validateInput.value === "number") {
        isValid = isValid && validateInput.value >= validateInput.min;
        // console.log("min")
    }
    if (validateInput.max != null && typeof validateInput.value === "number") {
        isValid = isValid && validateInput.value <= validateInput.max;
        // console.log("max")
    }

    return isValid;
}
```

Autobind this key word

```
// Code goes here!
function AutoBind(_: any, _2: string, descriptor: PropertyDescriptor) {
    const originalMethod = descriptor.value;
    const adjDescriptor: PropertyDescriptor = {
        configurable: true,
        enumerable: false,
        get() {
            //this here refers to the object calling get()
            const boundFn = originalMethod.bind(this);
            return boundFn;
        }
    }
    //will override old descriptor
    return adjDescriptor;
}
class ProjectInput {
    //template with template brackets
    templateElement: HTMLTemplateElement;
    //where to put element
    hostElement: HTMLDivElement;
```

```
    //element with template brackets gone
    element: HTMLElement;
    titleInputElement: HTMLInputElement;
    peopleInputElement: HTMLInputElement;
    descriptionInputElement: HTMLInputElement;



  …

    @AutoBind
    private submitHandler(e: Event) {
        e.preventDefault();
        const userInput = this.gatherUserInput();
        if (Array.isArray(userInput)) {
            const [title, desc, people] = userInput;
            console.log(title, desc, people);
            this.clearInputs();
        }
    }

 …
}
```

Instantiate project list for active and finished state

```
//project list class
class ProjectList {
    templateElement: HTMLTemplateElement;
    hostElement: HTMLDivElement;
    element: HTMLElement;

    //use private key word to create var directly
    constructor(private type: "active" | "finished") {
        this.templateElement = document.getElementById("project-list")! as
HTMLTemplateElement;
        this.hostElement = document.getElementById("app")! as HTMLDivElement;
        const importedNode = document.importNode(this.templateElement.content, true);
        this.element = importedNode.firstElementChild as HTMLElement;
        this.element.id = `${this.type}-projects`;
        this.attach();
        this.renderContent();
```

```
    }

    private renderContent() {
        const listId = `${this.type}-projects-list`;
        this.element.querySelector("ul")!.id = listId;
        //insert the header
        this.element.querySelector("h2")!.textContent = this.type.toUpperCase() + "
PROJECTS";
    }

    private attach() {
        //append to </div> right before closing bracket
        this.hostElement.insertAdjacentElement("beforeend", this.element)
    }
}
```

```
const activeProjList = new ProjectList("active");
const finishedProjList = new ProjectList("finished");
```

Singleton pattern using class

```
//use singletons to manage project
class ProjectState {
    …

    //below 3 lines for singleton structure
    //private - can only access within class
    //static - can only access by static method
    //so only static method within class can access instance below
    private static instance: ProjectState;
    //constructor is only accessible within class, so you cannot call new
ProjectState();
    private constructor() {

    }
    //getInstance will only return the same instance
    static getInstance() {
        if (this.instance) {
            //will only return the same instance - singleton
            return this.instance;
        }
```

```
        //we can use constructor within the class due to private constructor
        this.instance = new ProjectState();
        return this.instance;
    }

    …
}

//guarantee always have 1 instance,  when you get instance again, you will get the
same instance
const projectState = ProjectState.getInstance();
//constructor is private only acccessible within the class
// const projectStateInstan = new ProjectState();
```

Pass project state from projectState to ProjectList

- projectList class create a variable “assignedProjects” to contain the projects
- When instant the class, create a function
    - 1) (projects)=>{this.assignedProjects = projects} //pass projects from projectstate
    - 2) render these projects in UI

```
//project list class
class ProjectList {
    templateElement: HTMLTemplateElement;
    hostElement: HTMLDivElement;
    element: HTMLElement;
    //varaible for containing projects created by users
    assignedProjects: any[];

    //use private key word to create var directly
    constructor(private type: "active" | "finished") {
        this.templateElement = document.getElementById("project-list")! as
HTMLTemplateElement;
        this.hostElement = document.getElementById("app")! as HTMLDivElement;

        this.assignedProjects = [];

        const importedNode = document.importNode(this.templateElement.content, true);
        this.element = importedNode.firstElementChild as HTMLElement;
        this.element.id = `${this.type}-projects`;
```

```
        //when instance init, add listen function to get the projects and also render
them
        projectState.addListener((projects: any[]) => {
            this.assignedProjects = projects;
            this.renderProjects();
        })

        this.attach();
        this.renderContent();
    }

    //find UI element and render projects inside ui
    private renderProjects() {
        const listEl = document.getElementById(`${this.type}-projects-list`)! as
HTMLUListElement;
        for (const projItem of this.assignedProjects) {
            const listItem = document.createElement("li");
            listItem.textContent = projItem.title;
            listEl.appendChild(listItem);
        }
    }
}
```

ProjectState

- Receive listener function from above (assigned Projects = projects), whenever added new project, execute the function to pass the state over

```
//use singletons to manage project
class ProjectState {
    private listeners: any[] = []
    private projects: any[] = [];

    //below 3 lines for singleton structure
    //private - can only access within class
    //static - can only access by static method
    //so only static method within class can access instance below
    private static instance: ProjectState;
    //constructor is only accessible within class, so you cannot call new
ProjectState();
    private constructor() {

    }
```

```
    //getInstance will only return the same instance
    static getInstance() {
        if (this.instance) {
            //will only return the same instance - singleton
            return this.instance;
        }
        //we can use constructor within the class due to private constructor
        this.instance = new ProjectState();
        return this.instance;
    }

    //a list of array of fn
    addListener(listenerFn: Function) {
        this.listeners.push(listenerFn);
    }


    addProject(title: string, description: string, numOfPeople: number) {
        const newProject = {
            id: Math.random().toString(),
            title: title,
            description: description,
            numOfPeople: numOfPeople
        }
        this.projects.push(newProject);
        //loop thru f listener to execute on our projects []
        for (const listenerFn of this.listeners) {
            listenerFn(this.projects.slice())
        }
    }
}

//guarantee always have 1 instance,  when you get instance again, you will get the
same instance
const projectState = ProjectState.getInstance();
```

Redefine types throughout the project

```
enum ProjectStatus { active, finished };

//use class Project to pre project tyle
```

```
class Project {
   constructor(
       public id: string,
       public title: string,
       public description: string,
       public people: number,
       public status: ProjectStatus
   ) { }
}

//project statement management
type Listener = (items: Project[]) => void;
```

```
class ProjectState {
   private listeners: Listener[] = []
   private projects: Project[] = [];


…
addProject(title: string, description: string, numOfPeople: number) {
       const newProject = new Project(
           Math.random().toString(),
           title,
           description,
           numOfPeople,
           ProjectStatus.active
       )
       this.projects.push(newProject);
       //loop thru f listener to execute on our projects []
       for (const listenerFn of this.listeners) {
           listenerFn(this.projects.slice())
       }
   }
```

Filter project by its status

- Use Enum set up to filter projects by instance.type
- Clear inner html of UI, so that appendChild does not keep pushing in repeated project item

```
//project list class
class ProjectList {
```

```
    …

    //use private key word to create var directly
    constructor(private type: "active" | "finished") {
        …

        //when instance init, add listen function to get the projects and also render
them
        projectState.addListener((projects: Project[]) => {
            const relevantProjects = projects.filter(proj => {
                if (this.type = "active") {
                    //if 0===0
                    return proj.status === ProjectStatus.active;
                }
                return proj.status === ProjectStatus.finished;
            });
            this.assignedProjects = relevantProjects;
            this.renderProjects();
        })

        this.attach();
        this.renderContent();
    }

    //find UI element and render projects inside i
    private renderProjects() {
        const listEl = document.getElementById(`${this.type}-projects-list`)! as
HTMLUListElement;
        //clear UI inner content cos we repeatedly append child
        listEl.innerHTML = "";
        for (const projItem of this.assignedProjects) {
            const listItem = document.createElement("li");
            listItem.textContent = projItem.title;
            listEl.appendChild(listItem);
        }
    }
```

Created abstract class so that we can inherit from projectInput and projectList

```
abstract class Component<T extends HTMLElement, U extends HTMLElement>{
templateElement: HTMLTemplateElement;
   hostElement: T;
   element: U;

   constructor(
       templateId: string,
       hostElementId: string,
       insertAtStart: boolean,
       newElementId?: string,
   ) {
       this.templateElement = document.getElementById(templateId)! as
HTMLTemplateElement;
       this.hostElement = document.getElementById(hostElementId)! as T;

       const importedNode = document.importNode(this.templateElement.content, true);
       this.element = importedNode.firstElementChild as U;
       if (newElementId) {
           this.element.id = newElementId;
       }
       this.attach(insertAtStart);
   }

   private attach(insertAtStart: boolean) {
       this.hostElement.insertAdjacentElement(insertAtStart ? "afterbegin" :
"beforeend", this.element)
   }
   //you cannot have private abstract method
   abstract configure(): void;
   abstract renderContent(): void;
}
```

Then we extends from child class
ProjectList class

```
class ProjectList extends Component<HTMLTemplateElement, HTMLDivElement> {

   assignedProjects: Project[];

   constructor(private type: "active" | "finished") {
```

```
        super("project-list", "app", false, `${type}-projects`);

        this.assignedProjects = [];
        this.configure();
        this.renderContent();
    }

    configure() {
        projectState.addListener((projects: Project[]) => {
          …
    }

    renderContent() {
        const listId = `${this.type}-projects-list`;
        this.element.querySelector("ul")!.id = listId;
        //insert the header
        this.element.querySelector("h2")!.textContent = this.type.toUpperCase() + "
PROJECTS";
    }

    //find UI element and render projects inside i
    …
}
```

We did the same thing to extends to projectInput class

Use generics for creating listener function

```
//create generic type of array parameter, becos we donot know the type in the
beginning
type Listener<T> = (items: T[]) => void;

//state class is a generic type with T, T in the class can be customized when extended
below
class State<T>{
    protected listeners: Listener<T>[] = [];
    constructor() { }
    addListener(listenerFn: Listener<T>) {
        this.listeners.push(listenerFn);
    }
```

```
}

class ProjectState extends State<Project>{
   private projects: Project[] = [];
   private static instance: ProjectState;
   private constructor() {
       super();
   }
…
```

React with TS

Use below “.” to directly create react app in a folder without additional folder layer

npx create-react-app . --typescript

TSX extension

Ts with JSX code together

React.FC - functional component which must return jsx code

```
import React from 'react';

const App: React.FC=()=> {
 return (
   <div className="App">
     <h1>hello world</h1>
   </div>
 );
}

export default App;
```

Define a component with props - props must be defined

TodoList.tsx

```tsx
import React from "react";
interface TodoListProps{
   items: {id: string, text: string}[]
}

//must define the props explicitly
const TodoList: React.FC<TodoListProps>=(props)=>{
   return <ul>
       {props.items.map(t=><li key={t.id}>{t.text}</li>)}
   </ul>
}
export default TodoList;
```

Useref to hook on input element

```tsx
import React,{useRef} from "react";

const NewTodo: React.FC=()=>{

   //useRef on input element, must define your ref and init with null
   const textInputRef = useRef<HTMLInputElement>(null);

   const formSubmitHandler=(event: React.FormEvent)=>{
       event.preventDefault();
       const textInput = textInputRef.current!.value;
       console.log(textInput)
   }

   return <form onSubmit={formSubmitHandler}>
       <div>
           <label htmlFor="input-text">input text</label>
           <input type="text" id="input-text" ref={textInputRef}/>
       </div>
       <button type="submit">ADD TODO</button>
   </form>
}

export default NewTodo;
```

Use call back f from children back to parent

App.tsx

```
...

const App: React.FC=()=> {
 const todos = [{id: "ti", text:"finished the course"}]

 const todoAddHandler=(text:string)=>{
   console.log(text);
 }
 return (
   <div className="App">
     {/* you already defined the props, so auto completion here */}
     <NewTodo onAddTodo={todoAddHandler}/>
    …
   </div>
 );
}

export default App;
```

NewTodo.tsx

```
import React,{useRef} from "react";

type newTodoProps={
   onAddTodo:(text: string)=>void;
}

//define the props with the props
const NewTodo: React.FC<newTodoProps>=props=>{
   const textInputRef = useRef<HTMLInputElement>(null);

   const formSubmitHandler=(event: React.FormEvent)=>{
       event.preventDefault();
       const textInput = textInputRef.current!.value;
       //use the props f
       props.onAddTodo(textInput);
   }

   return <form onSubmit={formSubmitHandler}>
       …
   </form>
}
```

```
export default NewTodo;
```

UseState with ts

todo.model.ts

```
export interface Todo {
   id: string;
   text: string;
}
```

App.tsx
useState for array must define the structure of the array
We import from our external type interface

```
import React from 'react';
import TodoList from './components/TodoList';
import NewTodo from './components/NewTodo';
import {Todo} from "./todo.model";

const App: React.FC=()=> {
  const [todos, setTodos] = React.useState<Todo[]>([])

  const todoAddHandler=(text:string)=>{
    //setState function takes in prevState and update it with new ones
    setTodos(prevState=>[...prevState,{id: Math.random().toString(), text: text}])
  }
  return (
    <div className="App">
      {/* you already defined the props, so auto completion here */}
      <NewTodo onAddTodo={todoAddHandler}/>
      <TodoList items={todos}/>
    </div>
  );
}

export default App;
```

Set up on delete props

App.tsx

Set up todoDeleteHandler and pass to todoList.tsx

```
…

 const App: React.FC=()=> {
   const [todos, setTodos] = React.useState<Todo[]>([])

   …

   //set up onDelete handler
   const todoDeleteHandler=(id: string)=>{
     setTodos(prevState=>{
       return prevState.filter(t=>t.id!==id);
     })
   }
   return (
     <div className="App">
       <NewTodo onAddTodo={todoAddHandler}/>
       <TodoList items={todos} onDelete={todoDeleteHandler}/>
     </div>
   );
 }

 export default App;
```

todoList.tsx

- Redefine its props
- Use the onDelete to pass the id to be filtered on the new state

```
import React from "react";
interface TodoListProps{
   items: {id: string, text: string}[];
   //define onDelete props
   onDelete: (id: string)=>void;
}

//must define the props explicitly
const TodoList: React.FC<TodoListProps>=(props)=>{
   return <ul>
       {props.items.map(t=>
       <li key={t.id}>{t.text}<button
```

```
        // use onDelete props
        onClick={()=>{props.onDelete(t.id)}}
        >DELETE</button></li>)}
    </ul>
}
export default TodoList;
```

Other react ts

- Redux comes with doc to support ts
- For react router, no document yet, but when you install react router, you can install below
- Npm install - -save-dev @types/react-router-dom (add support ts for react router dom)