Typescript

類別 class

Class type

- public 為預設，可不寫
- private 私有屬性 子類別無法使用父類別的 private
- protected 可授權子類別使用，又能保有私有性質 敏私有
- # JS的私有自段 硬私有
- readonly 唯讀 只允設定一次，重新賦值會error
- static 靜態屬性 可以使括號表示法也失敗，編譯後也無法訪問
- abstract 抽象屬性
  - 抽象類別 class 不用 new就能直接使用，例如：Math.PI, ...
  - 抽象方法 function 類別無法建立實體(不能new)，必須建立子類別使用
    - 父定義此'方法的名稱與屬性'
    - 子類別定義'方法內容'(必須)
- 建定義的變數/參數成為可選的 ? 變數 GGG?: number;
  - 參數 constructor(VVV: string, DDD?: string)(...)
  - 或加入預設值 預設值為：undefined QQQ: number = 12;
    QQQ?: number = 12;

s.key // ts檢查 error
s["key"] // 括號表示法會通過 ts檢查 ts會檢查，但編譯後仍可從外部訪問到 s.key // ts檢查 error

繼承 Inheritance

- 初始化 constructor 初始化簡寫
- super(); 使用 extends 繼承父類別
  - 調用父類別的初始化 constructor()
  - 對子類別的初始化，需在super()後
- 屬性 父類別的 private無法使用
  - 父類別的 protected可使用
- 功能：可擴充、可覆寫 覆寫需注意要遵循被覆寫項的類別

```
class Base { greet() { ... } }
class Derived extends Base {
  greet(name: string) { ... } // error
  greet(name?: string) { ... } // 用?可選，可以合法覆寫
}
```

getter · setter

- 可設定驗證，過濾權限，過濾輸入的內容
- 使類別可像使用屬性一樣使用
- TS特殊規則 如果 get存在但不存在 set，則屬性自動 readonly

Singletons & Private Constructors

- 確保只能被創建一個的方法
  - 建立靜態屬性，type為自身類別
  - constructor 加上 private(無法被外界調用)
  - 建立一個靜態 function(功能為創建類別且檢測只創建一次)

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

介面/界面/接口 Interface

intro

- 比較 與 Type Aliases 很像
  - 擴展 type Animal = { name: string }
    interface Age { name: string }
    type Bear = Animal & Age & { honey: boolean } & type
    interface Animal { name: string }
    interface Bear extends Animal { honey: boolean } extends interface
  - 擴展數量限制 type與 interface皆可同時擴展/合併多個(class繼承父類別一次只能從一個)
  - 修改
    - type 無法更改
    - interface 可向原介面添加新字段

```
interface Window { title: string }
interface Window { ts: TypeScriptAPI }
```

- 因為以前 type沒那麼多功能/自由/靈活，才出現 Interface
- 為純 TS語法，在編譯過程中檢查代碼後不編譯出來，編譯後的js檔不會有 Interface
- https://www.typescriptlang.org/docs/handbook/2/objects.html

write

```
interface intro {
  name: string;
  introduction(phrase: string): void;
}
```

- 擴展/合併
  - 使用 extends interface BBB extends AAA {}
  - 可同時擴展/合併多個 interface DDD extends EEE, RRR, ... {}
- 資料型態
  - readonly 唯讀 readonly name: string; 只能設定一次，重新賦值會error 套用在類別時，類別中未寫readonly，也能限制無法重新賦值
  - 定義為可選 ? 變數 name?: string;
    函數 info?(text: string): void;
    參數 info(text?: string): void;

use

- 屬性、類型必須與Interface定義的一樣(?為可填)
- 可擴充使用更多 Interface的定義 (class繼承父類別一次只能兒一個)
- 可以在不知道具體樣貌時，保證有 Interface 指定的功能可使用 class XXX implements OOO, YYY, ZZZ {}
- 可自由新增 Interface 以外的屬性、方法，指限制必須包含哪些功能，不限制新增功能 let user: intro; --> user一定有 intro定義的功能可使用
  user = new Emp(...);

for class

```
interface intro {
  name: string;
  introduction(phrase: string): void;
}
```

- write class Emp implements intro {}
- use let adt: AddFn = (n1, n2) => {...};

for function

- write 補充：type 的寫法 interface AddFn { (a: number, b: number): number; } type AddFn = (a: number, b: number) => number;
- use

for object

配置

- tsconfig 文檔 https://www.typescriptlang.org/docs/handbook/tsconfig-json.html
- 編譯器配置文檔 https://www.typescriptlang.org/docs/handbook/compiler-options.html
- VS Code TS 調試 https://code.visualstudio.com/docs/typescript/typescript-debugging

第三方資源

DTS聲明文件：XXX.d.ts

- 若安裝的第三方是以 js寫的，ts認不得將無法 import，需經過"聲明文件"翻譯後 ts才能使用 有的套件安裝時已內建DTS，若沒有可嘗試搜尋翻譯包 ex: npm i —save @types/lodash
- 最終 ts都會編譯成 js，DTS文件只在開發過程中協助轉換 所以使用 "npm i -D"安裝即可
- declare
  - declare const AAA: string;
  - 若找不到翻譯包的最後手段，強制 ts認識這個字

lib

- JS 套件
  - ex: Lodash npm i —save lodash
  - npm i —save @types/lodash 另外搜尋 DTS文件，使 TS可正常使用 JS套件
- 已內建 for ts的 DTS 已內建 DTS文件 ex: class transformer
- TS 套件 無需 DTS文件 ex: class validate
- 練習：Google地圖 API -> 參考第13章

模塊 Module

namespace

- 導出 namespace App { export ... }
- 導入 namespace App { ... } /// <reference path="components/project-list.ts" /> 三斜線：為寫給TS的語法，reference：從 path引用檔案

ES模塊 ES-Module

- 導出 export ...
- 導入
  - 導入指定項 import { ProjectList } from "./components/project-list.js";
  - 使用別名導入 import { ProjectList as PJL } from "./components/project-list.js"; new PJL("active");
  - 導入全部 import * as PJL from "./components/project-list.js"; new PJL.ProjectList("active");
- default
  - 導出 export default ...
  - 導入 可重新取別名 import PJl from "./components/project-input.js";

更多關於 ES 模塊： https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

JavaScript 模塊（概述）：https://medium.com/computed-comparisons/commonjs-vs-amd-vs-requirejs-vs-es6-modules-2e814b114a0b

拆分檔案成多個文件

易於管理/開發

- 解法：使用打包工具 編譯成多個檔案，造成瀏覽器多次的請求(request)，效能較差
- 解法：將檔案編譯成一個檔，開啟 tsconfig.json中 "outFile": "./dist/bundle.js" 此僅給 TS認檔案去檢查，編譯成JS後會出現錯誤，錯誤原因為 namespace是 TS特有，編譯後不存在，編譯後檔案分散儲存 (具體原因不明白，但還要改 "module": "AMD"才能使用)，把 dist檔案刪光重新編譯就可以看到只編譯出 bundle.js一個檔了呦

功能匯入

- npm install —-save-dev ...
- webpack
- webpack-cli
- webpack-dev-server
- ts-loader webpack 可以用各種 loader 來處理項任務，其中 ts-loader 可以處理 typescript 的任務

webpack.config.js

- 新增預設打包檔 import 的路徑上不用再加".js"，Webpack會自動查找
- 入口設置：開始應用程序打包過程的起點 entry: "..."
  - 默认为 ./src
  - entry: "./src/app.ts"
- output: { ... }
  - 輸出設置
  - filename: "bundle.js" 打包成什麼檔
  - publicPath: "/dist" 打包後的路徑 webpack-dev-server 打包檔案為緩存狀態，並未真實儲存在路徑上（打開 dist資料夾下是看不到的
  - path: path.resolve(__dirname, "dist") 所有輸出文件的目標路徑 必須是絕對路徑
- module: { ... } 模塊配置相關
- rules: [{ ... }]
  - test: /\.ts$/ .ts結尾的檔要打包
  - use: "ts-loader" 用"ts-loader"的方式打包
  - exclude: /node_modules/ 這個不要包
- devtool: ... 開發工具：控制 是否/如何 生成 source map
  - "inline-source-map" 建立.map檔(方便debug)
  - false 包成最小的檔案(天書)
- resolve: { ... } extensions: [".ts", ".js"] 使用的擴展名 module在 import檔案時，路徑結尾不用寫要什麼檔，告訴我找哪些我一次幫你找

設置

- webpack.config.prod.js 新增不同設置的打包檔

打包指令

- "build": "webpack" 使用預設打包檔
- "bulid": "webpack --config webpack.config.prod.js" 指定打包檔指令
- "start": "webpack server" 建立指令 建立虛擬伺服器

使用

官方 Webpack 文檔： https://webpack.js.org/

型別 type

原始型別

- number 1 8.3 -2
- string '' "" ``
- boolean true false
- null
- undefined

any *

Array

- string[] class: ['Horror', 'mystery']
- Array<number> >參考 泛型 Generics

Tuple [number, string] menu: [1, "Now I'm opening!"]

object {} saleInfo: { prise: 399, inventory: 37 }

聯合類型 Union Types | (string | number) >參考 關於型別-型別檢查

類型(別名) Type Aliases type type User = { name: string; age: number | string }; 需要多次使用的對像類型(object)和聯合類型(Union)，可以編寫為 type，且取個好辨識的別名

接口/介面 interface interface interface Animal { name: string } >參考 介面/界面/接口 Interface

Enum {}

- enum EnterID { Customer, System, Viewer }
- enum CustomerLv { "A" = "Gold", "B" = "Silver", "C" = "Copper" }

function type

- 輸入參數聲明 function intro(name: string) { ... }
- 輸出類型聲明
  - function getNumber(): number { return ... }
  - void 函數不返回值 為TS自動推斷出的類型 注：不返回任何值的函數將隱式返回值 undefined，但是在 TypeScript中 void與 undefined不同
  - Unknown 這類似於 any類型，但此值做任何事情是不合法的
  - never 函數不返回值 函數拋出異常或終止程序的執行
- 匿名函數 TS會依上下文自動判斷參數類型 const names = ["Alice", "Bob", "Eve"]; names.forEach((s) => {...}) => s被 TS判別為 string

https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

關於型別 More About Types

表達式

- ?. 可選鏈接運算符 Optional Chaining operator 當引用或函數可能是 undefined或 null，不會跳 error product?.info?.name
- ?? 空值合併運算符 Nullish Coalescing operator 相當於先驗證存在後才尋找下一項 product && product.info && product.info.name
  - 和 || 功用相同，但只有 null或 undefined視為 false || 空字串與0也視為false
  - 優先級
    - 低於 ||
    - 高於 三元
    - 不能直接與 && 和 || 結合使用，需加上括號

型別檢查/限縮 Type Guard

注：TS上的 & 與 |

- & : Intersection 更像是邏輯閘的AND -- 且
  - 用在 object 結果：object 的結合 type SSS = FFF & NNN;
  - 用在聯合型別 結果：共有的 type type A_ID = number | string; type B_ID = number | boolean; type C_ID = A_ID & B_ID; // C_ID 的 type是 number
- | : Union 更像是邏輯閘的OR -- 或
  - 用在 object 結果：二/多 選一 type SSS = FFF | NNN;
  - 用在 class 結果：二/多 選一 type SSS = FFF | NNN;

JS上的 & 與 | 是二進制運算符

判斷型別

- typeof 判斷原始型別 typeof(fff) === 'number'
  不能判斷 Type Aliases (類型別名) 判斷不出自訂名稱，全是 object
- in 從內容物判斷 if('aaa' in fff)
  object, interface, class 都可使用
- instanceof JS的方法，用於驗證從哪個 class 建立出來的實體(new) if(fff instanceof FFF)
  能用在 class，但不能分辨 type 跟 interface
- type(類型別名) 或 interface 設定時，強制一項公共的屬性，即可由此屬性做區分 不必由內容物判斷，也不用 class new建立，處理已知的屬性，TS還會檢查錯字

```
interface Tree { type: '樹'; }
interface Flower { type: '花'; }
```

```
switch (plant.type) {
  case '樹': ...
  case '花': ... }
```

型別轉換 Type Casting

- 前面加上：<TYPE> const htmlDom = <HTMLInputElement> document.getElementById('user-input')!;
- 後面加上：as TYPE const htmlDom = document.getElementById('user-input')! as HTMLInputElement;

```
(htmlDom as HTMLInputElement).value = 'Input something here ...'
```

索引類型 Index Properties

```
interface ErrorContainer {
  [prop: string]: string;
}
```

當不知道類型屬性的所有名稱，可以限制名稱的形狀 名稱限制只能為 string或 number

函式超載 Function Overloads

對函式/方法的輸入與輸出強制進行型別註記

```
function Add(a: number, b: number): number;
function Add(a: string, b: string): string;
function Add(a: number, b: string): string;
function Add(a: string, b: number): string;
```

通常輸入有被複合過後的型別，TS會自動推論出至少兩種以上的型別，所以函式中需要型別檢查，但TS的型別推論並不會包含函式中寫的型別檢查結果

```
function Add(a: string | number, b: string | number) {  // 型別推論輸出為 string | number
  if (typeof a === 'string' || typeof b === 'string') // 型別檢查
    return a.toString() + b.toString();
  return a + b;
}
const R = Add(10, ' 3'); // R 的型別推論為 string | number
R.split('s');  // .split為 string的方法，若沒有 overload強制註記型別，string | number的型別 TS會警告，編譯不會error
```

https://www.typescriptlang.org/docs/handbook/2/types-from-types.html

泛型 Generics

將型別化名進行參數化，任何型別化名都可以轉換成泛用型別

- 函式 function
  - 類別 Class
  - 型別 Types type SituationPuzzle<T> = { [prop: string]: T }
  - 介面 Interface interface TagList<T> { tag: T | null; }

```
class DDD<T extends number | string | boolean> {
  private data: T[] = [];
  constructor(public value: Y[]) { ... }
  addItem(item: T) { ... }
  removeItem(item: T) { ... }
  getItem() { return this.data; }
}
```

```
function merge(objA: object, objB: object) { // 輸出型別判斷為 object
  return Object.assign(objA, objB);
}
const mergeObj = merge({ name: 'Frank' }, { age: 33 });
console.log(mergeObj);     // mergeObj型別判斷為 object
console.log(mergeObj.age);  // 但不知道此 object具體組成，會 error => 使用泛型 =>
```

```
function merge<T, U>(objA: T, objB: U) { // 輸出型為：T & U
  return Object.assign(objA, objB);
}
const mergeObj = merge({ name: 'Frank' }, { age: 33 }); // 型別判斷為 {name: string} & {age: number}
console.log(mergeObj);
console.log(mergeObj.name); // 型別判斷 {name: string, age: number} name屬性為已知，故不會error
```

約束 extends 需求舉例：function只要 input有 length屬性就能使用

- 若使用聯合型別(string| string[]...)可能漏寫有 length屬性的型別，也將無法使用後來自訂包含 length的型別
- 若使用函示超載(overload)也有疏漏的可能，code也超長
- 使用泛型可以更靈活，不用在乎具體的型別，只要約束(extends) input有 length屬性

```
interface Lengthy { length: number; }
function YYY<T extends Lengthy>(x: T): [T, string] {
  if (x.length === 1) { ... } // 指定輸入有 length屬性可使用，不會error
  return ...;
}
```

keyof 告訴 TS需要確保輸入有正確的結構，讓在訪問 object下的屬性時不會出錯

```
function RRR<T extends object, U extends keyof T>(obj: T, key: U) {
  return obj[key];  // input確認結構，訪問 object下的屬性時就不會出錯
}
```

實用程序類型 Utility Types

TS提供常見的"類型"轉換小工具 https://www.typescriptlang.org/docs/handbook/utility-types.html

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

```
interface PPP { AA: string; };
function creatLifeGoal(aa: number) {
  let ppp: Partial<PPP> = {}; // 加上 Partial使 PPP類型中的 AA變為可選(加上?)，所以建立空物件能不 error
  ppp.AA = "aa";
  return ppp as PPP;  // 有些TS版本用 Partial需要加上 as轉回原型別才能 return
}
```

- 將類型中全部屬性設置為可選的類型，將類別中全部屬性加上'?' function assign<T>(obj: T, props: Partial<T>): void;
- Readonly type Readonly<T> = { readonly [P in keyof T]: T[P] };
  - 將類型轉成唯讀 const UUU: Readonly<string[]> = ["Love", "Money"]; UUU.push("Friend"); // error TS2339: Property 'push' does not exist on type 'readonly string[]'. => TS有 error，但編譯後能執行，輸出看 UUU 還是被改變了
    const LLL: Readonly<{ aaa: string }> = { aaa: 'aaa' }; LLL.aaa = 'ccc'; // error TS2540: Cannot assign to 'aaa' because it is a read-only property. => TS有 error，但編譯後能執行，輸出看 LLL.aaa 還是被改變了
  - vs 直接設屬性特徵 writable為 false TS的錯誤訊息一樣 error TS2540: Cannot assign to 'aaa' because it is a read-only property. 但屬性特徵 writable為 false時，執行時是直接console紅字，停止繼續執行 Uncaught TypeError: Cannot assign to read only property 'aaa' of object '#<Object>'

裝飾器 Decorators

裝飾器的用途在於，開發給予開發人員使用的工具，是一種設計模式的實現，現今很多第三方套件或框架，都在背後採用裝飾器 可以想像成一個 wrapper，一個用來包裝函式或類別的函式，它可以將傳入的函式或類別做特定的處理，最後再回傳修改之後的物件。透過 Decorator 可以達成在不修改原程式碼的狀況下，在執行原函式的前後做一些特定的操作，同時也把可以重複使用的邏輯切分出去。

運作順序

- 在'定義'時運作，不需要建立實體(new)
- 有多個裝飾器時：創建的 function的順序(由上至下讀取) => 執行裝飾器順序(由下至上執行)
  - ex1: 輸出 Decorator1... Decorator2... 裝飾器執行順序是由下往上執行的（Bottom-up)

```
function Decorator1(_: Function) { console.log('Decorator1...'); }  //1
function Decorator2(_: Function) { console.log('Decorator2...'); }  // 2
@Decorator2
@Decorator1
class ReadingDecorators2 { constructor() {}; }
```

  - ex2：有 Factory(裝飾器工廠) 輸出 Read2... Read1... First Next 創建的 function的順序(由上至下讀取) Read2() -> Read1() 執行裝飾器順序(由下至上執行) function (_: Function){ console.log(first) } -> function (_: Function){ console.log(next) }

```
function Read1(first: string) {
  console.log('Read1...');      // 2
  return function (_: Function) { console.log(first) };      // 3
}
function Read2(next: string) {
  console.log('Read2...');      // 1
  return function (_: Function) { console.log(next) };      // 4
}
@Read2('Next')
@Read1('First')
class ReadingDecorators { constructor() {}; }
```

裝飾器工廠 Decorator Factory

就是在裝飾器外再包一層 function，回傳的則是真正的裝飾器(導致運作順序看起來怪怪的原因)，這時外層接收的參數就可以在回傳的裝飾器中應用

```
function Logger2(logText: string) {
  return function (EEE: Function) { // 真正的 Decorator
    console.log(logText);
  };
}
@Logger2("Logging...")
class QQQ { constructor() {}; }
```

裝飾器種類

- Class Decorator 類別裝飾器 函式型別 (target: any) => void target 欄位所在 class名稱
- Method Decorator 方法裝飾器
  - constructor無效，可用在 get, set
  - 可以有回傳值
  - 函式型別 (target: any, propertyKey: string, descriptor: PropertyDescriptor) => void
    - target 方法/get/set 所在 class名稱
    - propertyKey 方法/get/set 名稱
    - descriptor PropertyDescriptor 屬性描述符 >參考 PropertyDescriptor 屬性描述符
- Property Decorator 欄位裝飾器
  - 可以有回傳值，但TS會忽略，無用
  - 函式型別 (target: any, propertyKey: string) => void
    - target 欄位所在 class名稱
    - propertyKey 欄位名稱
- Parameter Decorator 參數裝飾器
  - 加在參數前方
  - 可以有回傳值，但TS會忽略，無用
  - 函式型別 (target: any, propertyKey: string, parameterIndex: number) => void
    - target 使用此參數方法所在的 class名稱
    - propertyKey 使用此參數方法的名稱
    - parameterIndex 方法中此參數的排序，從0開始
    - 在 constructor上使用 propertyKey: undefined

https://oldmo860617.medium.com/十分鐘帶你了解-typescript-decorator-48c2ae9e246d

在class上以裝飾器替換或添加功能：ex:在 constructor增加功能(方法裝飾器無法用在constructor)，可以讓裝飾器在"建立實體"時運作(非"定義"時)，講的很簡略，需要時再看一遍查查其他資料 https://www.udemy.com/course/understanding-typescript/learn/lecture/16935728#content

https://www.typescriptlang.org/docs/handbook/decorators.html

PropertyDescriptor 屬性描述符

屬性特徵

- Properties 特性
  - value: any 存儲在屬性中的值：僅在未指定 get和 set函數時才能指定
  - writable: boolean 可寫入的：如果可以修改屬性設為 true，僅在未指定 get和 set函數時才能指定
  - enumerable: boolean 可枚舉的：物件內的屬性是否可以透過 for-in 語法來造代 Object中各屬性可分開設定，所以使用 for-in 遍例物件時，enumerable可以控制哪些屬性不顯示
  - configurable: boolean 可配置的：能否被刪除、修改writable、enumerable 及 configurable 設定配置，例外：configurable:false時，writable仍可從 true被改為 false
- Methods 方法
  - get(): any
  - set(v: any): void

屬性描述符只有 value能直接設定，其於皆須通過 Object.defineProperty 與 Object.definedProperties設定

- Object.defineProperty 設定單一個屬性的描述器
- Object.definedProperties 設定多個屬性

一般在使用屬性字面值宣告屬性的時候

- value: 代表屬性的值
- writable、configurable、enumerable 都為 true
- 無 get、set 設定

指令

- npm i —save
  - 為默認項，等同npm i
  - 使用什麼套件 package.json中列在"dependencies"之下
- npm i —save-dev
  - 簡寫：npm i -D
  - 只有在開發或測試的時候需要的套件 package.json中列在"devDependencies"之下

Typescript

- class 類別
- Interface 介面/界面/接口
- 指令 配置
- Module 模塊
- Webpack
- type 型別
- Advanced Types 高級型別
- Generics 泛型
- Decorators 裝飾器 PropertyDescriptor 屬性描述符