- Typescript
  - webpack
  - 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
  - type 型別
    - any *
    - null
    - undefined
    - number 1 8.3 -2 ISBN: 100098
    - string `` " " '' `` name: 'Read me now!'
    - boolean true false onSale: true
    - object {} saleInfo: { prise: 399, inventory: 37 }
    - Array string[] class: ['Horror', 'mystery']
    - Tuple [number, string] menu: [1, "Now I'm opening!"]
    - Enum {} enum EnterID { Customer, System, Viewer }
    - 類型(別名) Type Aliases type type User = { name: string; age: number | string };
    - 聯合類型 Union Types | (string | number)
    - function type
      - void
      - Unknown
      - never
    - https://www.typescriptlang.org/docs/handbook/2/everyday-types.html
  - Module 模塊
    - 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
    - namespace
      - 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
        - 解法：將檔案編譯成一個檔，開啟 tsconfig.json中 "outFile": "./dist/bundle.js"
        - (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一個檔了啦
      - 三斜線：為寫給TS的語法，reference：從 path引用檔案 /// <reference path="components/project-list.ts" />
      - 導入
        - namespace App { ... }
      - 導出
        - namespace App { export ... }
    - ES-Module ES模塊
      - 導出
        - export ...
      - 導入
        - import { ProjectList } from "./components/project-list.js";
        - import 使用別名導入
          - 導入指定項
            - 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
          - 可重命名
            - import PJI from "./components/project-input.js";
  - 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
    - readonly 唯讀 只能設定一次，重新賦值會error
    - static 靜態屬性
      - 不用new就能直接使用，例如：Math.PI, ...
      - 在class內部以class名稱取用(非this)
    - abstract 抽象屬性
      - 抽象類別 class
        - 類別無法建立實體(不能new)，必須建立子類別使用
      - 抽象方法 function
        - 父定義此'方法的名稱與屬性'
        - 子類別定義'方法內容'(必須)
    - 限定義的變數/參數成為可選的 ?
      - 變數
        - GGG?: number;
      - 參數
        - constructor(VVV: string, DDD?: string){}
        - 使用時可不傳?的參數
      - 或加入預設值
        - 預設值為：undefined
          - QQQ: number = 12;
          - QQQ?: number = 12;
  - class 類別
    - constructor 初始化
      - 初始化簡寫
    - Inheritance 繼承
      - 使用 extends 繼承父類別
      - super();
        - 調用父類別的初始化 constructor()
        - 對子類別的初始化，需在super()後
      - 功能：可擴充、可覆寫
      - 屬性
        - 父類別的 private無法使用
        - 父類別的 protected可使用
    - getter、setter 使類別可像使用屬性一樣使用
    - Singletons & Private Constructors 確保只能被創建一個的方法
      - 建立靜態屬性變數(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
  - Interface 介面/界面/接口
    - 用於 object 定義結構
      - 屬性、類型必須與Interface定義的一樣(?為選填)
    - 使用
      - 用於類別 – implements
        - class XXX implements OOO {}
          - 可以在不知道類別具體樣貌時，保證有 Interface 指定的功能可使用
        - class XXX implements OOO, YYY, ZZZ {}
          - 還可擴充更多屬性方法
        - 可自由新增 Interface 以外的屬性、方法
      - ?：讓定義成為可選的
        - 參數
        - 變數
        - 函數
    - 擴展/合併
      - 套用在類別時，類別中未寫readonly，也無法重新賦值 只能設定一次，重新賦值會error readonly 唯讀 資料型態
      - 使用 extends
        - interface BBB extends AAA {}
      - 可同時擴展/合併多個(class不能)
        - interface DDD extends EEE, RRR, ... {}
    - 自定義函數類型 function type
      - type 的寫法
        - type AddFn = (a: number, b: number) => number;
      - interface 的寫法
        - interface AddFn {
(a: number, b: number): number;
}
    - 介紹
      - 與 type 很像(可能另外畫比較圖)
      - 因為以前type沒那麼多功能/自由/靈活，才出現Interface
      - 為純typescript語法，再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
  - 表達式
    - 可選鏈接運算符 Optional Chaining operator ?.
      - product?.info?.name 當引用或函數可能是 undefined或 null，不會跳 error
      - product && product.info && product.info.name 相當於先驗證存在後才尋找下一項
    - 空值合併運算符 Nullish Coalescing operator ??
      - 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
    - 先認識TS上的 & 與 |
      - & : Intersection 更像是邏輯閘的AND – 且
        - type SSS = FFF & NNN; 結果：object 的結合 用在 object
        - type A_ID = number | string;
type B_ID = number | boolean;
type C_ID = A_ID & B_ID; // C_ID 的 type是 number 結果：共有的 type 用在聯合型別
      - | : Union 更像是邏輯閘的OR – 或
        - type SSS = FFF | NNN; 結果：二選一 用在 object
        - type SSS = FFF | NNN; 結果：二選一 用在 class
      - JS上的 & 與 | 是二進制運算符
    - 再了解如何判斷"二選一"是哪個"一"
      - typeof
        - typeof(fff) === 'number' 判斷 type
        - 判斷不出自訂名稱，全是 object 不能判斷自訂的 type 名稱
      - 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)
        - class 能用，不能分辨 type 跟 interface
      - 判斷 object 與 聯合型別(自訂名稱) 的小幫步
        - type 跟 interface 設定時，強制一項公共的屬性，用做區分
          - interface Tree { type: '樹'; }
interface Flower { type: '花'; }
        - 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
          - 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 ErrorContaniner {
[prop: string]: string;
}
    - 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，會出現至少兩種以上的型別推論，所以需要型別檢查，但TS的型別推論不會驗證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的型別會出錯
    - https ://www.typescriptlang.org/docs/handbook/2/types-from-types.html
  - Advanced Types 高級型別
  - Generics 泛型
    - 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
        - 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) { // 輸出型別為：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;} 有 name，故不會error
      - 類別 Class
        - class DDD<T extends number | string | boolean> {
private data: T[] = [];
constructor(public value: Y[]) { ... }
addItem(item: T) { ... }
removeItem(item: T) { ... }
getItem() { return this.data; }
}
      - 型別 Types type SituationPuzzle<T> = { [prop: string]: T }
      - 介面 Interface interface TagList<T> { tag: T | null; }
    - 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
        - interface Lengthy {
length: number;
}
function YYY<T extends Lengthy>(e: T): [T, string] {
if (e.length === 1) { // 指定輸入有length屬性可使用，不會error
...
}
return ...;
}
        - 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
      - 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
}
      - Partial 部分 加上 Partial在建立時可以空白{}，不會跳 error，有點像將第別中全部屬性改為選填(加上?)
        - interface PPP {
AA: number;
BB: string;
}
function creatLifeGoal(aa: number, bb: string) {
let ppp: Partial<PPP> = {}; // 加上 Partial在最初使用模板建立時可以空白{}，不會跳 error
ppp.AA = aa;
ppp.BB = bb;
return ppp as PPP; // 有些TS版本用 Partial需要加上 as轉回原型別才能 return
}
      - Readonly 唯讀 加上就唯讀啦
        - const UUU: Readonly<string[]> = ["Love", "Money"];
UUU.push("Friend"); // error
      - https ://www.typescriptlang.org/docs/handbook/generics.html
  - Decorators 裝飾器
    - 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)
        - function Decorator1(_: Function) { console.log('Decorator1...'); } //1
function Decorator2(_: Function) { console.log('Decorator2...'); } // 2
@Decorator2
@Decorator1
class ReadingDecorators2 { constructor() {}; }
        - ex1:
          - 輸出 Decorator1...
Decorator2... 裝飾器執行順序是由下往上執行的（Bottom-up）
        - ex2：有 Factory(裝飾器工廠)
          - 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() {}; }
          - 輸出
            - Read2... 創建的 function的順序(由上至下讀取)
            - Read1... Read2() -> Read1()
            - First 執行裝飾器順序(由下至上執行)
            - Next function (_: Function){ console.log(first) } -> function (_: Function){ console.log(next) }
      - 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)
    - Decorator Factory 裝飾器工廠 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() {}; }
    - Decorator 的種類
      - Class Decorator 類別裝飾器 函式型別 (target: any) => void
      - 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
          - PropertyDescriptor 屬性描述符
            - 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：如果在(propertyName in object)期間訪問該屬性設為 true
              - configurable: boolean 可配置的：能否修改配置
            - Methods 方法
              - get(): any
              - set(v: any): void
      - Property Decorator 欄位裝飾器
        - 可以有回傳值，但TS會忽略，沒屁用
        - 函式型別 (target: any, propertyKey: string) => void
      - Parameter Decorator 參數裝飾器
        - 加在參數前方
        - 可以有回傳值，但TS會忽略，沒屁用
        - 函式型別 (target: any, propertyKey: string, parameterIndex: number) => void
propertyKey是指使用此參數方法的名稱
在 constructor使用時propertyKey: undefined，target顯示的是class?
    - 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，可以讓裝飾器在"建立實體"時運作(非"定義"時)，實際操作講的很簡略，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