Software Studio
軟體設計與實驗

TypeScript

Hung-Kuo Chu
Department of Computer Science
National Tsing Hua University

CS2410

What is TypeScript?

- An open-source programming language developed by Microsoft.
- A JavaScript superset, with **static typing** support.
- Make app development **as quick and easy as possible.**

TypeScript in 5 minutes(tutorial)

Type systems

- There are two kinds of type systems in programming languages, **static** and **dynamic** typing.
- Static typing means compiler will do type checking when source code is being compiled.
- Dynamic typing will do type checking in runtime of a program.

Static typing: Pros and Cons

- Pros:
  - Better performance.
  - Easier to manage.
  - Prevent runtime error.
- Cons:
  - Usually hard to write/learn.
  - Need to compile before debugging.

Static typing example

```
int number;    // Define an integer variable

number = 1;
number = “Hello world!”    //ERROR
```

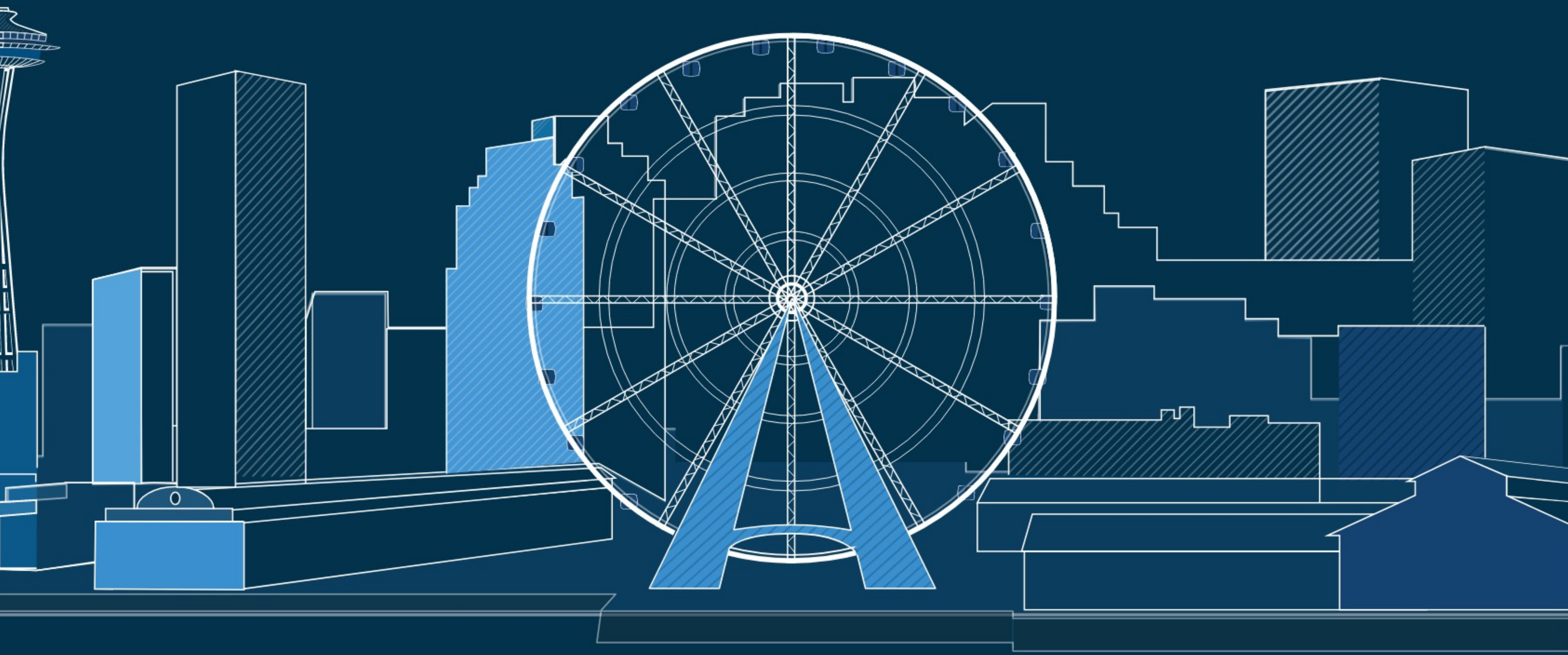
We will get error when assigning string to an integer.

Dynamic typing: Pros and Cons

- Pros:
  - – Usually easy to write/learn.
  - – Easier to declare a variable.
  - – No need to compile when debugging.
- Cons:
  - – Type error can cause runtime error.
  - – Hard to manage if code size is big.

Dynamic typing example

```
var number;    // Define a variable ‘number’

number = 1;      // ‘number’ is an integer
number = “Hello world!”  // ’number’ is a string
```



TypeScript

JavaScript that scales.

Why TypeScript?

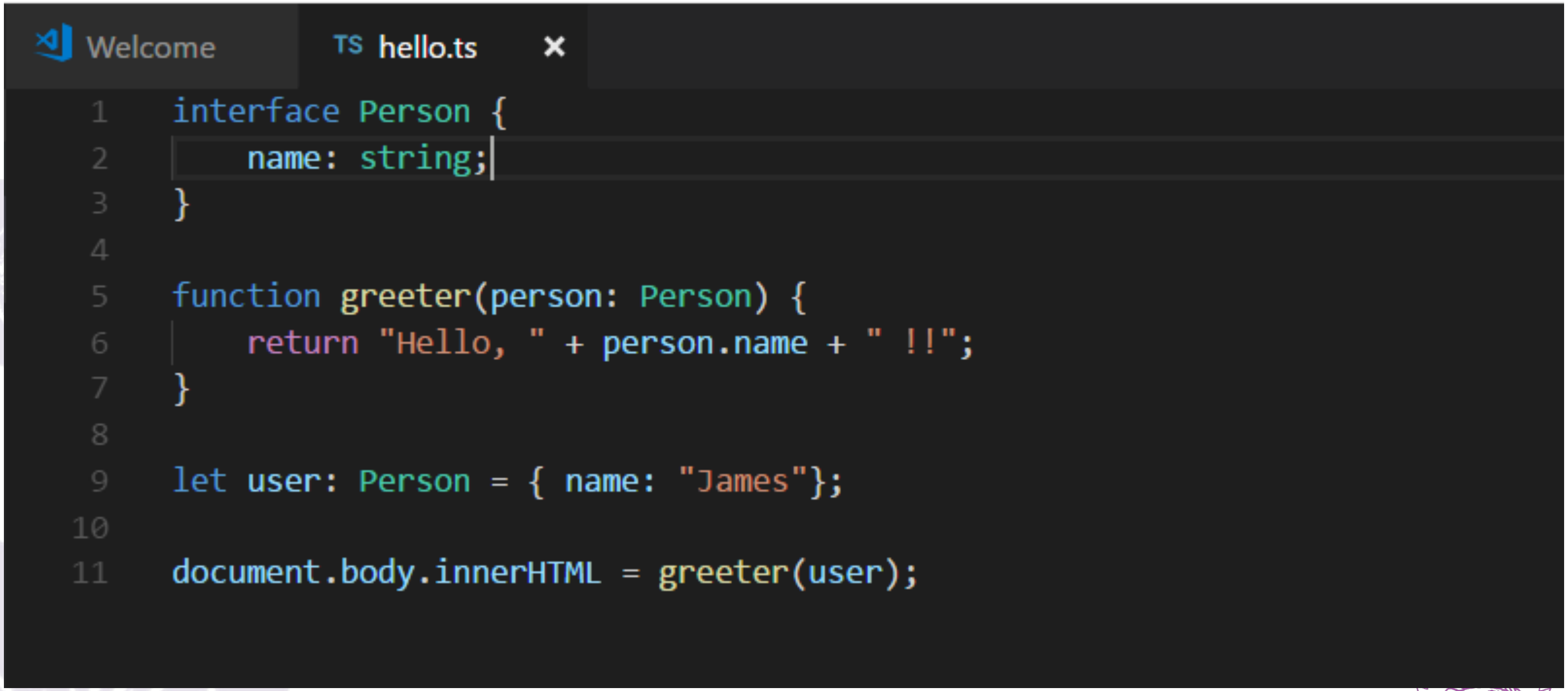
- **Type system** can enhance code quality and understandability.
- Provides **compile time** type safety for JavaScript code.
- Supports classes, interfaces and other **object-oriented programming techniques**.
- Try now in the TypeScript playground!

TypeScript Examples

```
//Define an interface named Person
interface Person {
    name: string;
}

//Define a function named greeter, with a parameter ‘Person’
function greeter(person: Person) {
    return "Hello, " + person.name + " !!";
}

//Define a variable user with Person type
let user: Person = { name: "James"};

document.body.innerHTML = greeter(user);
```

TypeScript with VSCode

VSCode supports TypeScript.
We can see syntax highlighting when editing.



```typescript
interface Person {
    name: string;
}

function greeter(person: Person) {
    return "Hello, " + person.name + " !!";
}

let user: Person = { name: "James"};

document.body.innerHTML = greeter(user);
```

Using TypeScript

- TypeScript is great, but it can’t be directly used in HTML documents.
- We will need a TypeScript **compiler** to translate TypeScript to JavaScript.
- Nowadays it is usually embedded inside the project’s build pipeline, used automatically.
- Refer to **Appendix-Create React App with TypeScript** to see how it is used inside a framework like React.

Installing TypeScript compiler separately

TypeScript: Basic Types

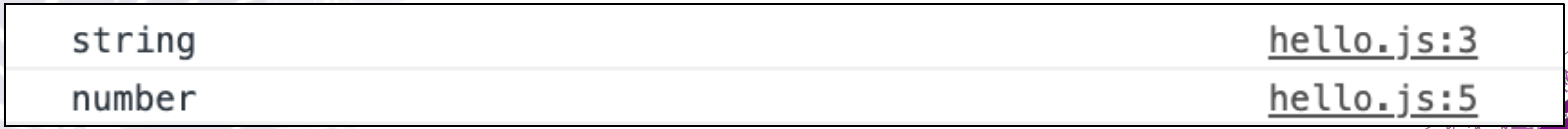
- In TypeScript, we can use ‘let’ to declare a variable with type.

```
let pi: number = 3.14;
let person: string = “James";
let sunnyDay: Boolean = false;
```

Type ‘any’

- If we don’t want to bind variable with a type, we can give it ’any’ type.

```
let i : any;

i = "A String!"
console.log(typeof i);

i = 12345;
console.log(typeof i);
```



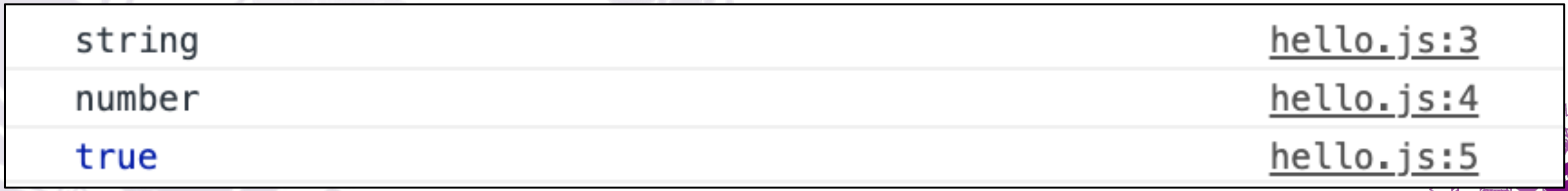
```
  string                                                      hello.js:3
  number                                                      hello.js:5
```

Practical example of using “any”

Syntax ‘typeof’

- If we want to know what type a variable is, we can use keyword ‘typeof’.

```
let i : string = "A string";
let j : number = 3.14159;

console.log(typeof i);
console.log(typeof j);
console.log(typeof i === "string");
```



```
string                                   hello.js:3
number                                   hello.js:4
true                                     hello.js:5
```

Type Aliases

- We can use keyword **‘type’** to define an alias of another type, like **typedef** in C. Note that ‘type’ will not create new type!

```
type Name = string;

let person1: string = "James";
let person2: Name = "Eric";
                                        “string”
console.log(typeof person1);
                                        “string”
console.log(typeof person2);
```

TypeScript: Function

- Functions in TypeScript provides:
  - Argument type checking.
  - Argument number checking.

Type of parameter is number

Return type is number

```
function add(first: number, second: number): number {
    return first + second;
}

console.log(add(1, 4));
console.log(add(3, “hello”));    // ERROR: string is not number
console.log(add(1, 2, 3));    // ERROR: Expected 2 arguments, but got 3.
console.log(add(1));    // ERROR: Expected 2 arguments, but got 1.
```

‘void’ Function

- Same as C/C++, functions in TypeScript can return nothing too.

```
function voidFunc1(): void{
    console.log("Returns Nothing!")
}

function voidFunc2(){
    console.log("Returns Nothing too!")
}
```

Advanced Type Checking

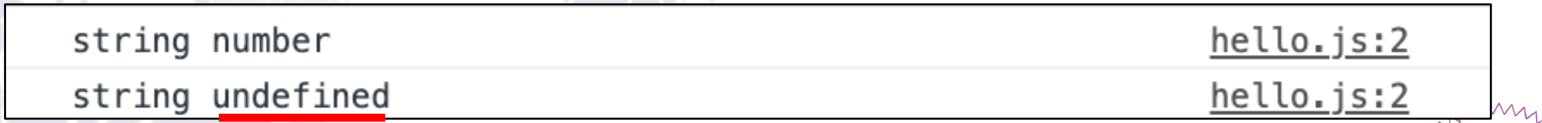
- We can use union type to check multiple types at the same time.

```
function hello(message: string | number) {
    //……
}

hello(100);     // OK
hello(‘Hello world!!’) // OK
```

Function Parameter

- We can bypass parameter number checking by adding ‘?’ in the right side of parameter name.

```
function saySomething(first: any, second?: any){
    console.log(typeof first, typeof second);
}
saySomething("123", 4);
saySomething(“567”);
```

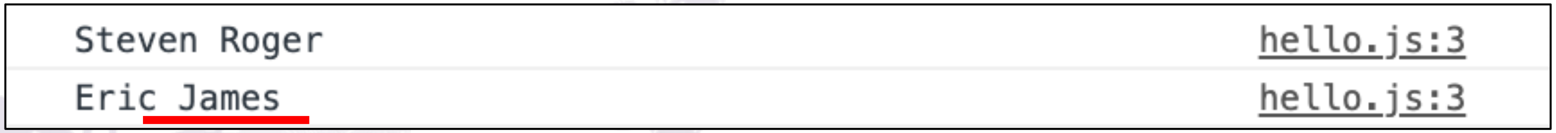


```
string number                                            hello.js:2
string undefined                                         hello.js:2
```

Default Parameter

- We can set default value for parameters.

```
function defFunction(name1: string, name2: string = "James") {
    console.log(name1, name2);
}

defFunction("Steven", "Roger");
defFunction("Eric");
```

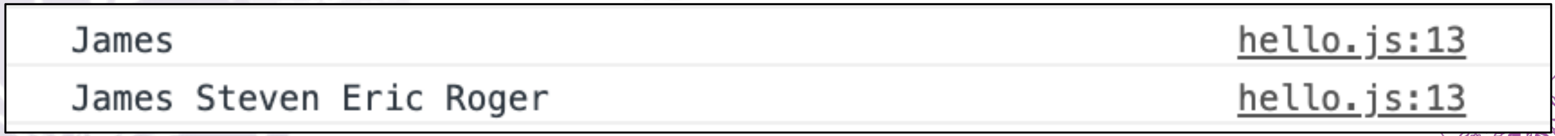


```
Steven Roger                                        hello.js:3
Eric James                                          hello.js:3
```

Rest Parameter

- When the parameters have the same type (ex. all strings), we can use the **rest parameter syntax (…)** to define a parameter with variable length (aka Array).

```
function memberName(leader: string, ...members: string[]) {
    console.log(leader + " " + members.join(" "));
}

memberName("James");
memberName("James", "Steven", "Eric", "Roger");
```

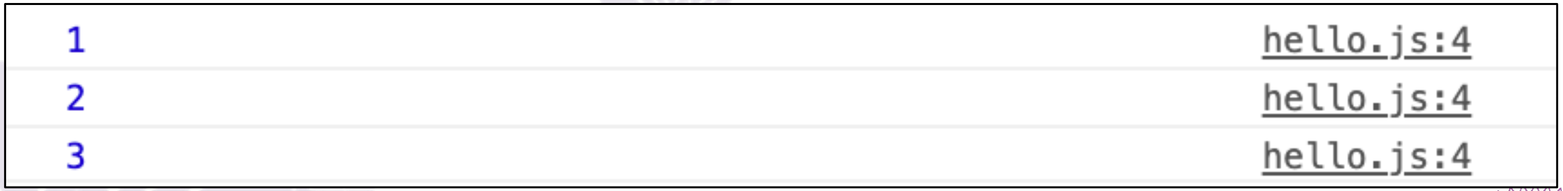


```
James                                             hello.js:13
James Steven Eric Roger                           hello.js:13
```

Iterator

- To iterate through a list or an array, we can use for loop.

```
let numbers = [1, 2, 3];

for (let num of numbers) {
    console.log(num);
}
```



```
1                                                  hello.js:4
2                                                  hello.js:4
3                                                  hello.js:4
```

TypeScript enums

- Enums allow us to define a set of named constants.
- Using enums can make it easier to manage our source code.

```
enum Direction {
    Up = 1, // Assigned explicitly to be 1 (Default 0)
    Down, // Implicitly 1+1 = 2
    Left, // Implicitly 2+1 = 3
    Right, // Implicitly 3+1 = 4
}
```

TypeScript enums (Cont’d)

- We can also use string in enums to define string constants.

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```

TypeScript Class

- We can use keyword ‘class’ to define a TypeScript class.

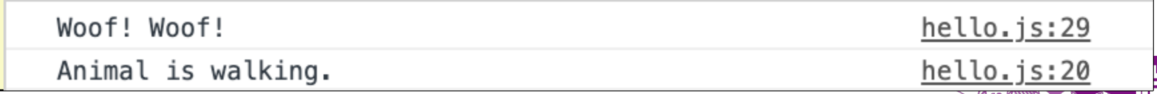
We will need keyword ‘this’ to access member variables.

```
class Person {
    public name: string;
    public id: number;
    public getInfo() {
        console.log("Name: " + this.name + " ID: " + this.id.toString());
    }
}

let p1: Person;
p1.name = "James"; p1.id = 1;
p1.getInfo();
```

Class Inherit

- To inherit from base class, we can use keyword ‘extends’.

```
class Animal {
    move() {
        console.log(“Animal is walking.”);
    }
}
class Dog extends Animal {
    bark() {  console.log(‘Woof! Woof!’);  }
}
const dog = new Dog();
dog.bark();
dog.move();
```



```
Woof! Woof!                                        hello.js:29
Animal is walking.                                 hello.js:20
```

Access Modifiers

- Using access modifiers can specify the accessibility of a class member.
- There are three types of access modifier in TypeScript:
  - **Public**: Access is not restricted. (Default)
  - **Private**: Only accessible inside the class.
  - **Protected**: Only accessible inside this class and its child class.

Access Modifiers Example

```
class Person {
    private id;
    protected name;
    public greet() {
        console.log(this.name + " say hello!")
    }
}
class Student extends Person {
    public greet() {
        console.log(this.id)    // Not accessible
        console.log(this.name)  // Accessible
    }
}
let std: Student = new Student();
std.greet(); // Accessible
console.log(std.name); // Not accessible
console.log(std.id); // Not accessible
```

TypeScript Interface

- We can use interface to define a prototype of a type, including member field and functions.
- It doesn’t provide implementation or initialization.

```
interface Point2D{
    x: number;
    y: number;
}

let origin: Point2D = {x: 0, y: 0};
```

Interface 用法

Class and Interface

- A class can implement one or multiple interfaces with the ‘implements’ keyword.

```
interface Named {
    name: string;
}
interface Identified {
    id: number;
}
class Student implements Named, Identified{
    public name: string; // Compile error if this line be removed
    public id: number; // Compile error if this line be removed
}
```

Class and Interface

- Interfaces provide **structural typing** to TypeScript.
- An object is considered to have implemented an interface if it has **every property defined in the interface**.

Class and Interface

```
interface Point2D {
  x: number;
  y: number;
}
class Point3D{ // Point3D implicitly implements Point2D
  public x: number = 0;
  public y: number = 0;
  public z: number = 0;
}
let origin: Point3D = new Point3D();
// Remove x or y from Point3D and this won't compile
let points: Point2D[]  = [origin];
```

Abstract Class

- We can define an abstract class that restricts the classes that extend it using the **abstract** keyword.
- Abstract classes can define implementations, but **a class cannot extend more than one abstract class**, unlike **interfaces**.
- Abstract classes cannot be instantiated.

Abstract Class Example

```
abstract class Character{
    protected _hp: number = 10;
    public get hp(): number { return this._hp }
    abstract onZeroHp(): void; // Called once when this._hp <= 0.
    public damage(val: number): void {
        if(this._hp > 0){
            this._hp -= val;
            if(this._hp <= 0) this.onZeroHp();
        }
    }
    attack(other: Character, damageVal: number){
        // Implementations of “Character” can use this method to attack
        // other characters.
        other.damage(damageVal);
    }
}
```

Abstract Class Example (Cont’d)

```
class Player extends Character{
     onZeroHp(): void{
         console.log(“Game over!”);
     } // …
}
class Enemy extends Character{
    protected scoreYield: number = 100;
    onZeroHp(): void{
        console.log(“Added score: ” + this.scoreYield);
    } // …
}
```

TypeScript Modules

- TypeScript shares the module system from ES6.
- Every .ts file can be seen as different **modules** that contain various **declarations (variables, classes, functions, etc.)**, like headers in C.
- A module can **export** its declarations for other modules to use and **import** declarations from other modules as well.
- TypeScript modules are named after their file paths **without the .ts at the end.**

Modules 說明書

TypeScript import/export

```
// Math.ts
export function greatestCommonDivisor(a: number, b: number){
  if(a < b) return greatestCommonDivisor(b, a);
  if(a == 0) return b;
  return greatestCommonDivisor(b, a % b);
}
export function leastCommonMultiple(a: number, b: number){
  return Math.abs(a * b) / greatestCommonDivisor(a, b);
}
```

```
// Main.ts
import {greatestCommonDivisor, leastCommonMultiple} from “./Math”;
// Main.ts can now call the two functions in Math.ts
```

TypeScript import/export

- You can export multiple declarations in one line.

```
// Math.ts
// The curly brackets {} are needed even if you only have one declaration.
export {greatestCommonDivisor, leastCommonMultiple};
function greatestCommonDivisor(a: number, b: number){
  // …
}
function leastCommonMultiple(a: number, b: number){
  // …
}
```

TypeScript export all

- You can export every declaration in a module with the **export * as** syntax.

```
// Math.ts
export * as Math;
// Referenced as “Math.greatestCommonDivisor” externally.
function greatestCommonDivisor(a: number, b: number){
  // …
}
// Referenced as “Math.greatestCommonDivisor” externally.
function leastCommonMultiple(a: number, b: number){
  // …
}
```

TypeScript import all

- Similarly, you can import every export a module has.

```
// Main.ts
import * as Math from “./Math”;
// Main.ts can now call the two functions in Math.ts, under the Math
namespace.
```

TypeScript import/export

- You can also rename imports or exports for convenience.

```
// Math.ts
export {greatestCommonDivisor as gcd, leastCommonMultiple as lcm}
function greatestCommonDivisor(a: number, b: number){
  // …
}
function leastCommonMultiple(a: number, b: number){
  // …
}
```

```
// Main.ts
import {gcd as g, lcm as l} from “./Math”;
// Main.ts now uses “g” to refer to “gcd” and “l” to refer to “lcm”
```

TypeScript ‘default’ export

- Each module can also have a default export that makes the import/export syntax more concise.

```
// Character.ts
export default abstract class Character {
  // (Possible implementation in Page 35)
}
```

```
// Game.ts
import Character from “./Character”;
// Without “default”, you will need to add curly brackets {} around Character.
```

Modules and Namespaces

- TypeScript also has **Namespaces** before ES6 introduced modules.
- Their usage is similar to namespaces in C++ and can also be imported/exported in the module system, but **we do not recommend it.**
- In modern TypeScript, it is recommended to use **modules** over namespaces.

TypeScript: Namespaces and Modules

Modules and Namespaces

```
// Constants.ts
export default namespace Constants {
    const pi: number = 3.14159;
    const e: number = 2.71828;
}
```

```
// Main.ts
import Constants from “./Constants”
console.log(“pi = ” + Constants.pi + “ and e = ” + Constants.e);
```

References

- TypeScript home page
- TypeScript tutorial
- TypeScript GitHub page
- 讓 TypeScript 成為你全端開發的 ACE！
- Typescript 初心者手札

thank you!
Question?