TRƯỜNG ĐẠI HỌC
SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH
HCMC University of Technology and Education
HCMUTE

KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN CÔNG NGHỆ PHẦN MỀM

CÔNG NGHỆ PHẦN MỀM MỚI (MTSE431179)

Ngôn ngữ TypeScript

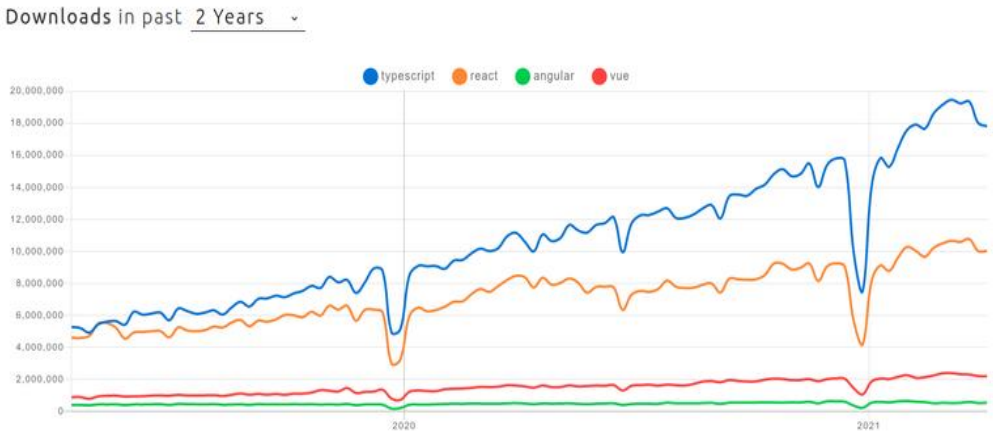
THS. NGUYỄN HỮU TRUNG

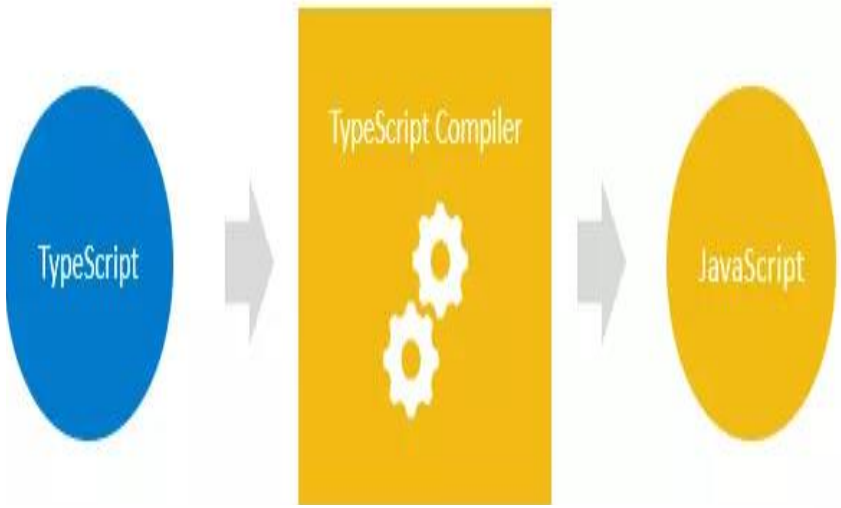
Thông tin Giảng viên

- Ths. Nguyễn Hữu Trung
- Khoa Công Nghệ Thông Tin
- Trường Đại học Sư Phạm Kỹ Thuật TP.HCM
- 090.861.7108
- trungnh@hcmute.edu.vn
- https://www.youtube.com/@baigiai

Giới thiệu TypeScript

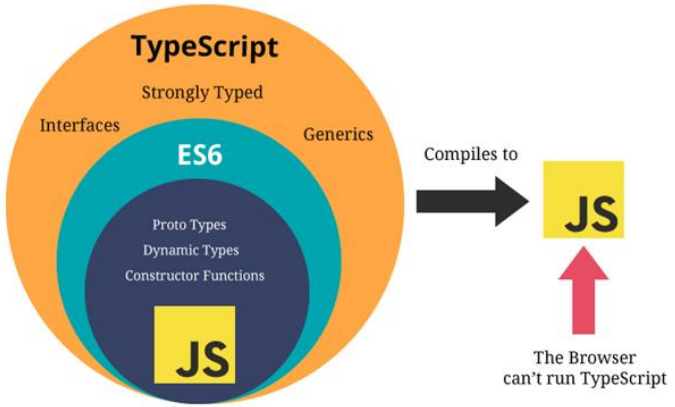
- TypeScript là một ngôn ngữ lập trình mã nguồn mở, được xây dựng dựa trên JavaScript. Phần mở rộng **.ts**





Stats

	stars	issues	updated	created	size
typescript	70,101	5,129	Apr 15, 2021	Jun 17, 2014	minzipped size 712.2 KB
react	166,896	696	Apr 15, 2021	May 24, 2013	minzipped size 2.8 KB
angular	59,596	464	Mar 31, 2021	Jan 6, 2010	minzipped size 62.3 KB
vue	181,993	542	Apr 13, 2021	Jul 29, 2013	minzipped size 22.9 KB



Cài đặt TypeScript

- **Node.js** : Node.js là môi trường.
- **TypeScript compiler**: Là một module Node.js sẽ biên dịch code TypeScript thành code JavaScript. npm install -g typescript / tsc --v
- **Visual Studio Code(VS code):** Là một editor hỗ trợ code TypeScript. **Live Server** : cho phép bạn khởi chạy một server local cho việc phát triển.

Khởi tạo Project TypeScript

- Bước 1: Tạo một folder để lưu code, ví dụ folder là: **vidu1**
- Bước 2: Chạy VS Code và mở folder đó.
  Bước 3: Tạo một file TypeScript gọi là **app.ts** với extension của file là **.ts**
- Bước 4: Thêm code bên dưới vào file app.ts

```
let message: string = 'Hello, World!';
console.log(message);
```

- Bước 5: Mở Terminal trong VS Code bằng keyboard shortcut **Ctrl+`** hoặc theo menu **Terminal > New Terminal**
- Bước 6: Biên dịch **tsc app.ts** và chạy **node app.js**

Type Annotations

- TypeScript sử dụng **type annotations**(chú thích kiểu dữ liệu) để chỉ định rõ ràng các kiểu dữ liệu cho các định danh như variables, function, objects, etc.
- TypeScript sử dụng cú pháp **: type** sau một định danh để làm type annotations, và **type** có thể là bất kỳ loại dữ liệu hợp lệ nào.

```
let variableName: type;
let variableName: type = value;
const constantName: type = value;
let arrayName: type[];
```

Ví dụ:

```
let name: string = 'John’;
let age: number = 25;
let active: boolean = true;
let names: string[] = ['John', 'Jane',
'Peter’];
let person: { name: string; age: number };
```

Number, String, Boolean trong TypeScript

- Tất cả các số trong TypeScript đều là giá trị floating-point(dấu phẩy động) hoặc số nguyên lớn. Các số dấu phẩy động có kiểu **number** trong khi số nguyên lớn có kiểu là **big int**.
- TypeScript cũng sử dụng cặp dấu nháy kép (") hoặc dấu nháy đơn (') để bao quanh các chuỗi ký tự
- Kiểu **boolean** cho phép 2 giá trị: **true** và **false**.

Ví dụ:

```
let name: string = 'John’;
let age: number = 25;
let active: boolean = true;
let names: string[] = ['John', 'Jane', 'Peter’];
let person: { name: string; age: number };
```

Array trong TypeScript

- Một mảng trong TypeScript là một danh sách dữ liệu có thứ tự: **let arrayName: type[];**

Ví dụ:

```
let names: string[] = ['John', 'Jane', 'Peter'];
```

```
let series = [1, 2, 3];
console.log(series.length);
```

```
let series = [1, 2, 3];
let doubleIt = series.map(e => e* 2);
console.log(doubleIt);
```

```
let scores : (string | number)[];
 scores = ['Programming', 5, 'Software Design', 4]; console.log(scores);
```

Enum trong TypeScript

- Enum là một nhóm tên các giá trị constant. Enum là kiểu liệt kê.

enum name {constant1, constant2, ...};

Ví dụ:

```
enum Month { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
```

```
function isItSummer(month: Month) {
    let isSummer: boolean;
    switch (month) {
    case Month.Jun:
    case Month.Jul:
    case Month.Aug:
        isSummer = true; break;
    default: isSummer = false; break;
    }
    return isSummer;
}
```

Void, Any TypeScript

- Kiểu **void** trong TypeScript được sử dụng để trả về loại của hàm, mà trong đó hàm không trả về bất kỳ giá trị nào

Ví dụ:

```
function log(message): void {
        console.log(messsage);
}
```

- Kiểu **any** cho phép bạn lưu trữ một giá trị thuộc bất kỳ kiểu nào.

```
// json may come from a third-party API
const json = `{"latitude": 10.11, "longitude":12.12}`;
// parse JSON to find location
const currentLocation = JSON.parse(json);
console.log(currentLocation);
```

```
let result: any;
result = 10.123;
console.log(result.toFixed());
result.willExist(); //
```

Type Assertions trong TypeScript

- Khi sử dụng Type Assertions, trình biên dịch sẽ coi một giá trị là một kiểu được chỉ định cụ thể. Sử dụng từ khóa **as**.

Ví dụ:

```
let netPrice = getNetPrice(100, 0.05, false) as number;
console.log(netPrice);

let netPrice = <number>getNetPrice(100, 0.05, false);
```

- Bạn cũng có thể sử dụng cú pháp ngoặc nhọn **<>** để xác nhận một kiểu, như sau: **<targetType>** value. Lưu ý rằng bạn không thể sử dụng cú pháp ngoặc nhọn **<>** với một số thư viện như React. Vì lý do này, bạn nên sử dụng từ khóa **as** cho các xác nhận kiểu dữ liệu

Interface trong TypeScript

- Interfaces trong TypeScript định nghĩa một tiêu chuẩn trong code của bạn. Chúng cung cấp các tên rõ ràng để kiểm tra loại dữ liệu. Một interface có thể có các thuộc tính tùy chọn. Để khai báo một thuộc tính tùy chọn, bạn sử dụng dấu (**?**) đặt ở cuối tên thuộc tính trong khai báo.

```
function getFullName(
        person: {
                firstName: string; lastName: string
        }) {
        return `${person.firstName} ${person.lastName}`;
}
let person = { firstName: 'John', lastName: 'Doe' };
console.log(getFullName(person));
```

```
interface Person {
        readonly ssn: string;
        firstName: string;
        lastName: string;
        middleName?: string;
}
```

```
function getFullName(person: Person) {
 return `${person.firstName} ${person.lastName}`;
}
let john = { firstName: 'John', lastName: 'Doe' };
console.log(getFullName(john));
```

Default parameters và Rest trong TypeScript

- JavaScript hỗ trợ các **default parameters** kể từ ES2015 (hoặc ES6) với cú pháp sau:

```
function name(parameter1=defaultValue1,...) {
// do something }
```

```
function applyDiscount(price, discount = 0.05)
{ return price * (1 - discount); }
console.log(applyDiscount(100)); // 95
```

- Một **Rest parameter**(các tham số còn lại) cho phép một hàm chấp nhận không hoặc nhiều đối số của kiểu được chỉ định. Trong TypeScript các rest parameter tuân theo các quy tắc sau:
  - Một hàm chỉ có một rest parameter
  - Rest parameter xuất hiện ở cuối danh sách các tham số
  - Loại của rest parameter là một loại mảng

```
function fn(...rest: type[]) { //... }
```

```
function getTotal(...numbers: number[]): number {
let total = 0; numbers.forEach((num) => total +=
num); return total; }
```

Functions trong TypeScript

- Khai báo

```
function name(parameter: type, parameter:type,...): returnType {
        // do something
}
```

```
function add(x: number, y: number): number { return x + y; }
let add2 = function (x: number, y: number): number { return x + y; };
console.log(add(10, 20)); //output 30
console.log(add2(10, 20)); //output 30
//Sử dụng với arrow function(mũi tên (=>) xuất hiện giữa các tham số và kiểu trả về.)
let add3 = (x: number, y: number) : number => { return x + y; }
let add4 = (x: number, y: number) => { return x + y; }
 let add5 = (x: number, y: number) => x + y;
let add6: (a: number, b: number) => number = function (x: number, y: number) {
return x + y; };
console.log(add3(10, 20)); //output 30
console.log(add4(10, 20)); //output 30
console.log(add5(10, 20)); //output 30
console.log(add6(10, 20)); //output 30
```

Function Overloadings trong TypeScript


```
function addNumbers(a: number, b: number):
number { return a + b; }
function addStrings(a: string, b: string): string {
return a + b; }
```



```
function add(a: number | string, b: number | string): number | string {
if (typeof a === 'number' && typeof b === 'number')
            return a + b;
 if (typeof a === 'string' && typeof b === 'string')
            return a + b;
}
```

Function Overloadings trong TypeScript

- Để mô tả tốt hơn các mối quan hệ giữa các kiểu được sử dụng bởi một hàm, TypeScript hỗ trợ **function overloadings**(nạp chồng hàm). Ví dụ

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
        return a + b;
}
```

```
function sum(a: number, b: number): number;
function sum(a: number, b: number, c: number): number;
function sum(a: number, b: number, c?: number): number {
        if (c) return a + b + c; return a + b;
 }
```

Function Overloadings trong TypeScript

- Khi một hàm là thuộc tính của một lớp, nó được gọi là một phương thức. TypeScript cũng hỗ trợ phương thức overloading

```
class Counter {
        private current: number = 0;
        count(): number;
        count(target: number): number[];
        count(target?: number): number | number[] {
                if (target) {
                        let values = [];
                        for (let start = this.current; start <= target; start++) {
                                values.push(start);
                         }
                this.current = target;
                return values;
        }
         return ++this.current;
} }
```

Class trong TypeScript

- Trong JavaScript thì không có khái niệm class như các ngôn ngữ Java, C#. Nhưng trong phiên bản ES5 bạn có thể sử dụng một hàm khởi tạo và kế thừa nguyên mẫu để tạo một class

```
function Person(ssn, firstName, lastName) {
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName;
}
```

- Bây giờ, bạn có thể định nghĩa một phương thức prototype để lấy full name của person như bên dưới.

```
Person.prototype.getFullName = function () {
    return `${this.firstName} ${this.lastName}`;
}
```

Tiếp theo, bạn có thể sử dụng class Person bằng cách tạo một new object:

```
let person = new Person('171-28-0926','John','Doe');
console.log(person.getFullName());
```

Class trong TypeScript

- Sử dụng class trong ES6, ví dụ:

```
class Person {
        ssn; firstName; lastName;
        constructor(ssn, firstName, lastName) {
                this.ssn = ssn;
                this.firstName = firstName;
                this.lastName = lastName;
} }
```

```
let person = new Person('171-28-0926','John','Doe');
console.log(person.getFullName());
```

```
class Person { ssn; firstName; lastName;
constructor(ssn, firstName, lastName) {
this.ssn = ssn; this.firstName = firstName; this.lastName = lastName;
}

getFullName() {
return `${this.firstName} ${this.lastName}`;
} }
```

Class trong TypeScript

- Class trong TypeScript sẽ thêm type annotations(chú thích kiểu) đến các thuộc tính và phương thức trong class

```
class Person {
        ssn: string; firstName: string; lastName: string;
        constructor(ssn: string, firstName: string, lastName: string) {
        this.ssn = ssn; this.firstName = firstName; this.lastName = lastName;
}
getFullName(): string { return `${this.firstName} ${this.lastName}`; } }
```

Access Modifiers trong TypeScript

- Access modifiers có nhiệm vụ thay đổi quyền truy cập các thuộc tính và phương thức trong **class**. TypeScript cung cấp 3 mức truy cập là:
  - **private** chỉ cho phép truy cập bên trong class đó
  - **protected** chỉ cho phép truy cập bên trong class đó và bên trong class kế thừa(class con)
  - **public** cho phép truy cập ở bất kỳ vị trị nào
- Chú ý: TypeScript sẽ kiểm soát truy cập trong thời gian biên dịch chứ không phải trong thời gian chạy.

```
class Person { private ssn: string; private
firstName: string; private lastName: string; //
... }
```

```
class Person { protected ssn:
string; // other code }
```

```
class Person { // ... public getFullName(): string { return
`${this.firstName} ${this.lastName}`; } // ... }
```

Getters và Setters trong TypeScript


```
class Person {
        public age: number;
        public firstName:
        string; public lastName: string;
}
```

Nhập dữ liệu

```
if( inputAge > 0 && inputAge < 200 ) {
        person.age = inputAge;
}
```

Để truy cập các thuộc tính của class **Person**, bạn có thể làm như sau:

```
let person = new Person();
person.age = 26;
```

Getters và Setters trong TypeScript


```
class Person {
        private _age: number;
        private _firstName: string;
        private _lastName: string;
public get age() { return this._age; }
public set age(theAge: number) {
        if (theAge <= 0 || theAge >= 200) {
                throw new Error('The age is invalid');
        }
        this._age = theAge;
}
public get firstName() { return this._firstName; }
public set firstName(theFirstName: string) { if (!theFirstName) { throw new
Error('Invalid first name.'); } this._firstName = theFirstName; }
public get lastName() { return this._lastName; }
public set lastName(theLastName: string) { if (!theLastName) { throw new
Error('Invalid last name.'); } this._lastName = theLastName; }
public getFullName(): string { return `${this.firstName} ${this.lastName}`; } }
```

Inheritance (kế thừa) trong TypeScript

- Một **class** có thể tái sử dụng các thuộc tính và phương thức của class khác. Cái này gọi là sự **inheritance** (kế thừa) trong TypeScript.
- Class kế thừa các thuộc tính và phương thức được gọi là **child class**(lớp con). Và class có các thuộc tính và phương thức được kế thừa được gọi là **parent class**(lớp cha).
- Sử dụng từ khóa **extends** để cho phép một lớp kế thừa từ một lớp khác
- Sử dụng **super()** trong constructor của lớp con để gọi constructor của lớp cha. Bạn cũng có thể sử dụng cú pháp **super.methodInParentClass()** để gọi **methodInParentClass()** trong phương thức của class con.

Inheritance (kế thừa) trong TypeScript


```
class Person {
        constructor(private firstName: string, private lastName: string) {
                this.firstName = firstName;
                this.lastName = lastName;
        }
        getFullName(): string {
                return `${this.firstName} ${this.lastName}`;
        }
       describe(): string {
                return `This is ${this.firstName} ${this.lastName}.`;
        }
}
```

```
class Employee extends Person { //.. }
```

Inheritance (kế thừa) trong TypeScript

Để gọi constructor của lớp cha trong constructor của lớp con, sử dụng cú pháp **super()**, ví dụ:

```
class Employee extends Person {
        constructor( firstName: string, lastName: string, private jobTitle: string)
{
                // call the constructor of the Person class:
                super(firstName, lastName);
        }
}
```

```
let employee = new Employee('John', 'Doe', 'Web Developer');
console.log(employee.getFullName());
console.log(employee.describe());
```

Abstract Classes trong TypeScript

- Một abstract class(lớp trừu tượng) là một lớp cơ sở, thường được sử dụng để định nghĩa các hành vi chung cho các lớp dẫn xuất(lớp kế thừa). Không giống như lớp bình thường, một lớp abstract không thể được khởi tạo trực tiếp.
- Để khai báo một lớp abstract, bạn sử dụng từ khóa **abstract** :

```
abstract class Employee { //... }
```

  - Các lớp trừu tượng không thể được khởi tạo.
  - Một lớp trừu tượng có ít nhất một phương thức trừu tượng.
  - Để sử dụng một lớp trừu tượng, bạn cần kế thừa nó và cung cấp thực hiện các xử lý cho các phương thức trừu tượng

Abstract Classes trong TypeScript

- Thông thường, một lớp abstract chứa một hoặc nhiều phương thức abstract .
- Một phương thức của lớp abstract không chứa các thực thi code bên trong. Nó chỉ định nghĩa tên của phương thức mà không thực thi gì. Các thực thi của phương thức abstract sẽ được thực hiện bên trong lớp dẫn xuất.

```
abstract class Employee {
constructor(private firstName: string, private
lastName: string) { }
abstract getSalary(): number
get fullName(): string { return
`${this.firstName} ${this.lastName}`; }
 compensationStatement(): string { return
`${this.fullName} makes ${this.getSalary()} a
month.`;
} }
```

Giải thích lớp **Employee**

- Constructor khai báo các thuộc tính **firstName** và **lastName**.
- Phương thức **getSalary()** là một phương thức abstract. Lớp dẫn xuất sẽ thực thi logic dựa trên loại của employee.
- Phương thức **getFullName()** và **compensationStatement()** chứa chi tiết các thực thi.
- Lưu ý rằng phương thức **compensationStatement ()** gọi phương thức **getSalary()**.

Generics trong TypeScript

- Sử dụng generics trong TypeScript cho phép bạn viết các hàm, class, interfaces có thể tái sử dụng và tổng quát hóa.

```
function getRandomElement<T>(items: T[]): T {
            let randomIndex = Math.floor(Math.random() * items.length);
 return items[randomIndex];
}
```

Modules Trong TypeScript

- Một module trong TypeScript có chứa cả khai báo và code. Một module thực thi trong phạm vi riêng của nó, không phải trong phạm vi toàn cục. Điều đó có nghĩa là khi bạn khai báo các biến, hàm, lớp, giao diện, v.v., trong một module, chúng không hiển thị bên ngoài module, trừ khi bạn xuất chúng một cách rõ ràng bằng cách sử dụng câu lệnh **export**
- Mặt khác, nếu bạn muốn truy cập các biến, hàm, lớp, v.v., từ một module, bạn cần nhập chúng bằng cách sử dụng câu lệnh **import**

```
import { Validator } from './Validator';
```

```
import * from 'module_name';
```

```
export interface Validator {
        isValid(s: string): boolean
}
```

```
interface Validator {
        isValid(s: string): boolean
}
export { Validator };
```

```
interface Validator { isValid(s: string): boolean }
export { Validator as StringValidator };
```