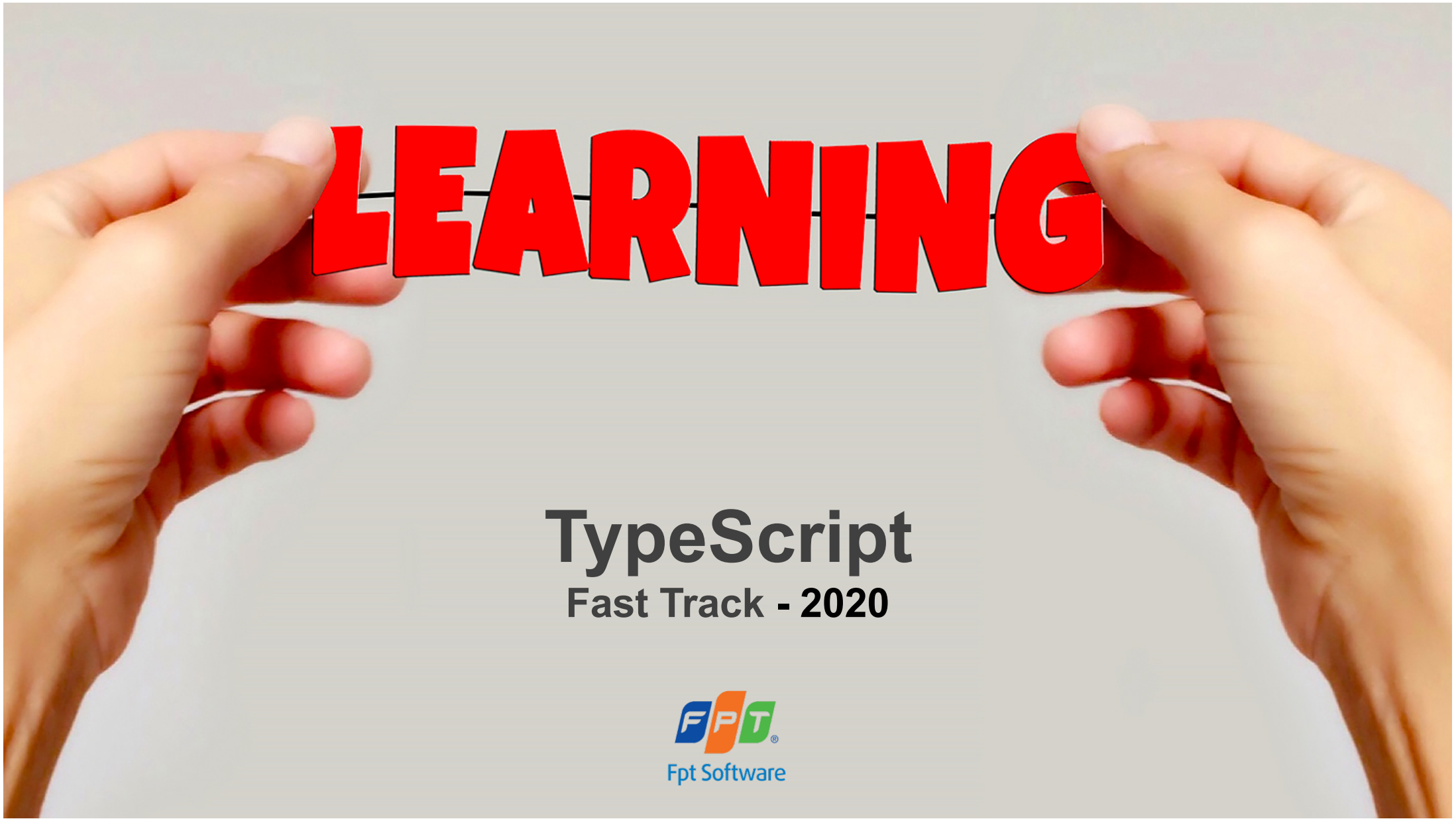


TypeScript

Fast Track - 2020

Fpt Software

Contents

1. Problem with JavaScript?
2. TypeScript
3. Types
4. Function
5. Class

Fpt Software

Contents

6. Interface

7. Generics

8.

9.

10.

Fpt Software

1. Problem with JavaScript ?

1. Problem

- Writing large applications in JavaScript is difficult
- Lacks static typing mechanism to help catch bug
- Lacks structuring mechanism like Class, Interface

```
// what is the output of following code?
function add(a, b) {
    return a + b;
}

console.log(add(2) / 3);
```

2. TypeScript

2. TypeScript Overview

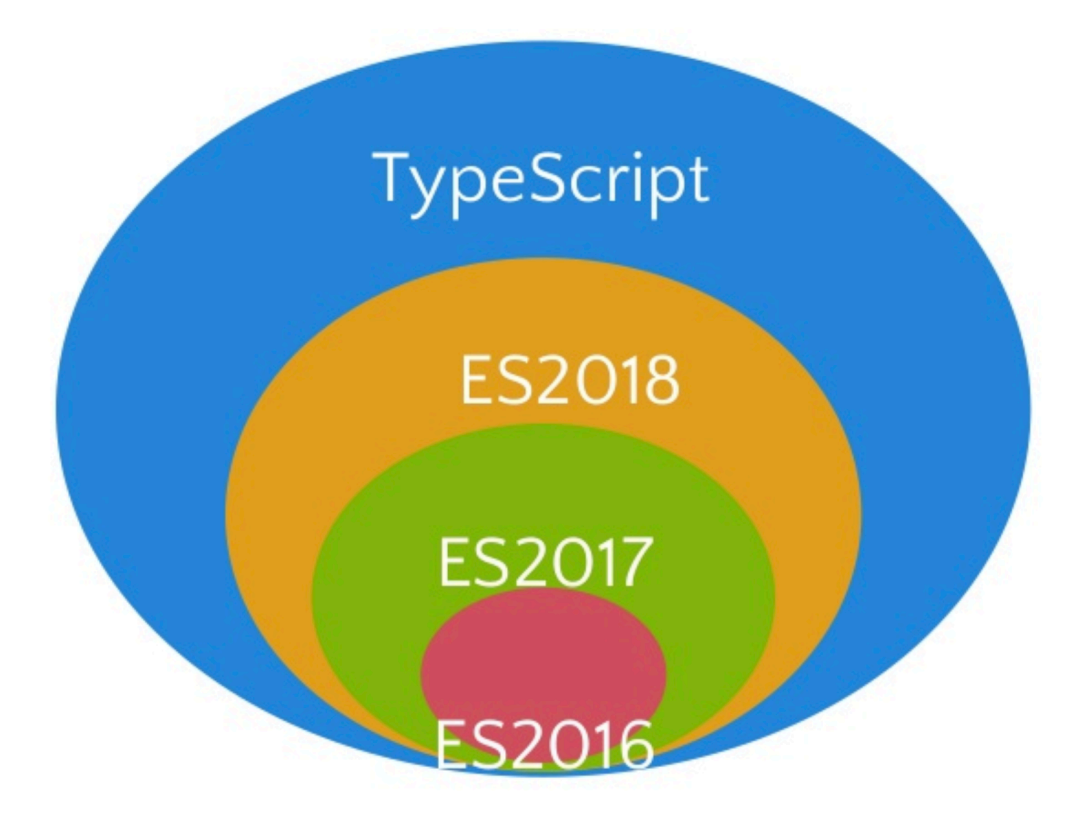
Fpt Software

- Helps in large scale JavaScript application
- Adds additional features like Static Type, Class, Interface
- Easy to convert from JavaScript to TypeScript
- Open Source
- Have all features of ES6

2. TypeScript and JavaScript

Fpt Software

- TypeScript is a super set of JavaScript with optionally typed



2. TypeScript History

- First made public in October 2012
- TypeScript 0.9, released in 2013, added support for Generics
- In July 2014, the development team announced a new TypeScript compiler, claiming 5x performance gains

2. TypeScript Features

Fpt Software

- Static Typing
- Optional Static Type Annotation
- Additional features for Functions
- Class
  - Field, property, method, constructor, static
- Interface
- Generics
- All features of ES6

2. Try TypeScript

Fpt Software

- Try TypeScript at: https://www.typescriptlang.org/play/

```typescript
class Vendor {
  name: string;

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

  greet() {
    return "Hello, welcome to " + this.name;
  }
}

const shop = new Vendor("Ye Olde Shop");
console.log(shop.greet());
```

```javascript
"use strict";
class Vendor {
    constructor(name) {
        this.name = name;
    }
    greet() {
        return "Hello, welcome to " + this.name;
    }
}
const shop = new Vendor("Ye Olde Shop");
console.log(shop.greet());
```

3. Types / Optional Type Annotation

3. Type Annotation

Fpt Software

- Annotate variables with types

```
1   const className: string = 'Fresher Angular';
2   const age: number = 20;
3   const isFresher: boolean = true;
4
```

- Show warning if there's a type mismatch (**!important**)

```
const isFresher: boolean
Type '"true"' is not assignable to type 'boolean'.
Peek Problem    No quick fixes available
const
const
const isFresher: boolean = 'true';
```

3. Type Annotation

- Type Annotation is author-time feature only. No additional code is emitted in the final JavaScript

```
1   const className: string = 'Fresher Angular';
2   const age: number = 20;
3   const isFresher: boolean = true;
4
5
6
7
```

```
1   "use strict";
2   const className = 'Fresher Angular';
3   const age = 20;
4   const isFresher = true;
5
6
7
```

3. Type Annotation

Fpt Software

- Basic Static Types
  - any
  - Primitive
    - boolean
    - number
    - string
    - void
    - null
    - undefined
  - Array

3. Type Annotation Syntax

- Syntax: `variable: <Data Type>`
- A colon after a variable name starts a *type annotation*: the *type signature* after the colon describes what values the variable can have
- Example:

```
var age: number = 32; // number variable


function display(id:number, name:string) {}
```

3. Primitive Data Types

- All numbers in TypeScript are floating point value
- Those floating point numbers get the type '**number**'

```
var x: number = 55;
var y: number = 123.4567;
```

- boolean – true/false value

```
var isFresher: boolean = true; // or false
```

3. Primitive Data Types

- string – single quote or double quote

```
var msg1: string = ‘Hello from Fresher’;
var msg2: string = ”Hello from Angular”;
```

- No char type;

```
var character = ‘a’; //what is the type of character ?
```

3. Optional Type Annotation

Fpt Software

- TypeScript tries to infer type

```
var x = 42;
```

```
var x: number
Type '"Fresher"' is not assignable to type 'number'.
Peek Problem    No quick fixes available
```

```
x = 'Fresher';
```

```
var fresher: {
    name: string;
    clazz: string;
}
var fresher = {
    name: 'Nguyen Van A',
    clazz: 'Angular'
}
```

3. Type Inference

- TypeScript tries to infer type **depend on how you declare variable**
- 4 ways of variable declaration:
  1. Declare its type and value in one statement
  2. Declare its type but no value
  3. Declare its value but no type
  4. Declare neither value nor type

3. Type Inference

Fpt Software

```
1   // Option 1: Declare type and value
2   var message1: string = 'Hello Fresher';
3
4   // Option 2: Declare type but no value
5   var message2: string;
6   // value can be assigned later
7   message2 = 'Fresher';
8
9   // Option 3: Declare value but no type
10  // TypeScrip will infer the type from value
11  var message3 = 'Angular';
12
13  // Option 4: Declare neither value nor type
14  // Type is inferred as any, value = undefined
15  var message4;
16
```

3. Type Array

- Syntax: `variables: <DataType>[]`

```
1    var cities: string[] = ['Hanoi', 'Hai Phong', 'Da Nang', 'Ho Chi Minh'];
2    var primes: number[] = [2, 3, 5, 7, 11];
3    var bools: boolean[] = [true, false, true, true];
4
```

- Every element of array must be of same Type

```
var primes: number[]
Type '"3"' is not assignable to type 'number'.
Peek Problem    No quick fixes available
primes[1] = '3';
```

3. Type Enum

- Addition to JavaScript datatypes
- Enum is a way to giving more friendly names to sets of values
- Syntax:

```
enum EnumName {
    Values1, Values2, Value3
}
```

3. Type Enum Example

Fpt Software

```
1  enum Color {
2      Red, Green, Blue
3  }
4
5
6  var blue = Color.Blue;
7  var red = Color.Red;
8
9  function checkColor(color: Color): void {
10     if (color == Color.Blue) {
11         console.log('Color is Blue');
12     }
13 }
14
```

3. Type any

Fpt Software

- Useful to describe unknown type of variables
- May come from dynamic content
- Allows to opt-out of type-checking
- Same as not declaring any datatypes

```
1    var notSure: any;
2
3    var list: any[] = [1, 2, '3', '4', true];
4    list[1] = 'Fresher';
5
```

3. Type void

- Opposite to ‘any’
- Describe the absence of having any type at all
- Commonly used as the return type of functions that do not return a value

```
function greet(s: string): void {
    console.log('Hello ' + s);
}
```

4. Function

4. Function Overview

Fpt Software

- Fundamental building block of any JavaScript application
- JavaScript supports Higher-Order Function
- Allows build up layers of abstraction
- Describe how to '**do**' thing
- TypeScript add new capabilities to standard JavaScript
  - Type Annotation for parameter and return type
  - Rest/Optional Default Parameter
  - Function overloads

4. Function

Fpt Software

- Allows parameter and return type annotation

```
1   function add(a: number, b: number): number {
2       return a + b;
3   }
4
5   function mul(a: number, b: number): number {
6       return a * b;
7   }
8
9   console.log(add(1, 2));
10  console.log(mul(add(1, 2), 3));
11  console.log(add('1', 3));
12
```

4. Function (2)

Fpt Software

- Show warning for type mismatch

```
1  function add(a: number, b: number): string {
2      return a + b + '';
3  }
4
5  function mul(a: number, b: number): number {
6      return a * b;
7  }
8
9  console.log(add(1, 2));
10 console.log(mul(add(1, 2), 3));
11
```

5. Class

5. Class Overview

- TypeScript is Object Oriented JavaScript
- Class is a blueprint for creating objects
- Class consists of
  - Fields to store data
  - Constructor to initialize fields
  - Methods to define behavior

5. Class Example

Fpt Software

```
1   class Person {
2       name: string; // declare name field of type string
3       age: number; // declare age field of type number
4
5       constructor(name: string, age: number) {
6           // initialize value for name and age field
7           this.name = name;
8           this.age = age;
9       }
10
11      greet(str: string): void {
12          console.log('Hello ' + str + ' from ' + this.name);
13      }
14  }
15
16
17  let p = new Person('Van A', 20);
18  p.greet('Van B');
19
20
```

5. Access Modifiers

- public (default)
- private

```
1  class Person {
2      private name: string;
3      private age: number;
4
5      constructor(name: string, age: number) {…
9      }
10
11     greet(str: string): void { // default to public method
12         console.log('Hello ' + str + ' from ' + this.getFullName());
13     }
14
15     // this method is only accessible inside the class Person
16     private getFullName(): string {…
18     }
19 }
20
21 let p = new Person('Van A', 20);
22 p.greet('Van B');
23
24 p.name;
25 p.getFullName();
26
```

5. Static

Fpt Software

- *static* member/method is visible on the class not on the instances

```
1   class Person {
2       private name: string;
3       private age: number;
4       static count = 0;
5
6   >   constructor(name: string, age: number) {…
11      }
12
13  >   static greet(str: string): void {…
15      }
16  }
17
18  let p = new Person('Van A', 20);
19  Person.greet('Van B');
20  Person.count;
21  p.greet('Van B');
22  p.id;
```

5. Inheritance

Fpt Software

- Inheritance of class through **extends** keyword

```
1  class Person {
2      private name: string;
3
4      constructor(name: string) {…
9      }
10
11     greet() {
12         console.log(this.name);
13     }
14 }
15
16 class Fresher extends Person {
17     private clazz: string;
18
19     constructor(name: string, clazz: string) {
20         super(name); // must call contructor of base class
21         this.clazz = clazz;
22     }
23
24     study() {
25         console.log('Study ' + this.clazz);
26     }
27
28 }
29
30 let anv = new Fresher('Nguyen Van A', 'Fresher Angular');
31 anv.greet();
32 anv.study();
```

5. Abstract Class

- Contains “abstract” method
- Cannot create an instance of abstract class

```
1  interface Flyable {
2      fly(): void;
3  }
4
5  abstract class Bird implements Flyable {
6      name: string;
7
8      constructor(name: string) {
9          this.name = name;
10     }
11
12     abstract fly();
13 }
14
15 let obj: Flyable = new Bird('Angry Bird');
16
17 obj.fly();
18
```

6. Interface

6. Interface

- Define a constraints or new type
- With “implements” keyword
- Can “extends” another interface

```
1  interface Flyable {
2      fly(): void;
3  }
4
5  class Bird implements Flyable {
6      fly() {
7          console.log('Bird');
8      }
9  }
10
11 class Eagle implements Flyable {
12     fly() {
13         console.log('Eagle');
14     }
15 }
16
17 let obj: Flyable = new Bird();
18
19 obj.fly();
20
```

6. Interface (2)

- Define a new type
- Optional field with **?** Keyword
- Show warning when access field that are not defined

```
1   interface UserData {
2       username: string;
3       password: string;
4       age?: number;
5   }
6
7   let data: UserData = {
8       username: 'anv',
9       password: 'anv'
10  };
11
12  data.name;
13
```

7. Generics

7. Problem

- We want to build a Stack of **number** ?

```
1   class Stack {
2       private data: number[];
3
4       constructor() {
5           this.data = [];
6       }
7
8       push(item: number) {
9           this.data.push(item);
10      }
11
12      pop(): number | undefined {
13          return this.data.pop();
14      }
15  }
16
```

```
17  let s = new Stack();
18  s.push(5);
19  s.push(1);
20  s.push(10);
21  console.log(s.pop()); // 10
22
```

7. Problem (2)

- Then we want to build a Stack of **string** ?

```
1   class Stack {
2       private data: string[];
3
4       constructor() {
5           this.data = [];
6       }
7
8       push(item: string) {
9           this.data.push(item);
10      }
11
12      pop(): string | undefined {
13          return this.data.pop();
14      }
15  }
16
```

```
17  let s = new Stack();
18  s.push('5');
19  s.push('1');
20  s.push('10');
21  console.log(s.pop()); // '10'
22
```

7. Problem (3)

- Then we want to build a Stack of **boolean** ?

```
1   class Stack {
2       private data: boolean[];
3
4       constructor() {
5           this.data = [];
6       }
7
8       push(item: boolean) {
9           this.data.push(item);
10      }
11
12      pop(): boolean | undefined {
13          return this.data.pop();
14      }
15  }
16
```

```
17  let s = new Stack();
18  s.push(true);
19  s.push(false);
20  s.push(true);
21  console.log(s.pop()); // true
22
```

7. Problem ?

- We solve the Stack problem but we create another bigger problem: code duplication

```
class Stack {
    private data: number[];

    constructor() {
        this.data = [];
    }

    push(item: number) {
        this.data.push(item);
    }

    pop(): number | undefined {
        return this.data.pop();
    }
}
```

```
class Stack {
    private data: string[];

    constructor() {
        this.data = [];
    }

    push(item: string) {
        this.data.push(item);
    }

    pop(): string | undefined {
        return this.data.pop();
    }
}
```

7. Generics

- Generics allow us to define **datatype** as variable

```
1   class Stack<T> {
2       private data: T[];
3
4       constructor() {
5           this.data = [];
6       }
7
8       push(item: T) {
9           this.data.push(item);
10      }
11
12      pop(): T | undefined {
13          return this.data.pop();
14      }
15  }
16
```

```
17  let stringStack = new Stack<string>();
18  stringStack.push('1');
19  stringStack.push('10');
20  stringStack.push('20');
21  console.log(stringStack.pop()); // 20
22
```

```
23  let numberStack = new Stack<number>();
24  numberStack.push(1);
25  numberStack.push(10);
26  numberStack.push(20);
27  console.log(numberStack.pop()); // 20
28
```

Happy Coding!

thank you!

Fpt Software