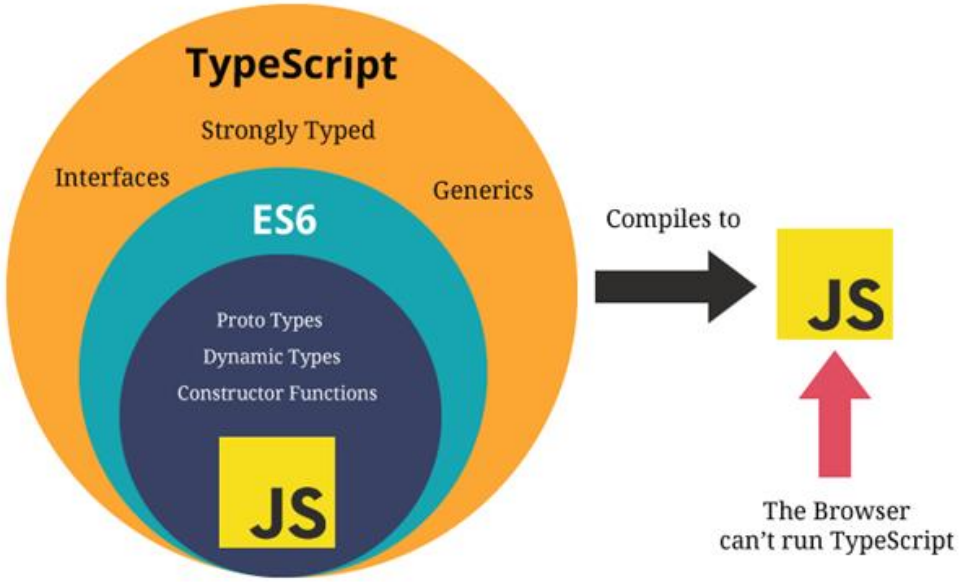
CS472 WAP TypeScript

Maharishi International University - Fairfield, Iowa

JavaScript vs. TypeScript

JavaScript is the most popular programming language on the web. It is used to create websites, servers, games, and native mobile apps. JavaScript is a loosely typed language.

TypeScript is a free and open-source high-level programming language developed by Microsoft (Oct 2012) that adds static typing with optional type annotations to JavaScript. It is designed for the development of large applications and transpiles to JavaScript.



Official website: https://www.typescriptlang.org

Source code: https://github.com/Microsoft/TypeScript

Why TypeScript?

One of the great things about type-checking is that:

1. It helps write safe code because it can **prevent bugs at compile time**.
2. Compilers can improve and run the code **faster**.

Types are **optional** in TypeScript *(because they can be inferred)*.

TypeScript Compiler

TypeScript compiles into simple JavaScript.

A TypeScript code is written in a file with `.ts` extension and then compiled into JavaScript using the TypeScript compiler.

Note: if you desire to transpile the ts files to js, you may use the following command: `npx tsc`

TypeScript Setup

Download and Install Node https://nodejs.org/en/download/

Install typescript

- `npm install -D typescript`

Check typescript version

- `tsc -v`

Compile your typescript code with TSC

- `tsc filename.ts`
- `tsc filename.ts –w //This is in watch mode`

Run your code with NodeJS

- `node filename.js`

tsconfig.json

Best practices, tsconfig.json is located in project root directory.

to generate a tsconfig.json file:

```
tsc --init
```

It used for

- Which files should be complied
- Which directory to compile them to
- Which version of JavaScript to emit
- …

tsconfig.json

```
{
    "compilerOptions": {

        "module": "commonjs",
        "outDir": "dist",
        "strict": true,
        "target": "ES2015"
    },
    "include": [
        "src"
    ]
}
```

- `include`: which folders should TSC look in to find your Type Script files?
- `module`: which module system should TSC compile your code to (CommonJS, ES2015, etc.)
- `outDir`: Which folder should TSC put your generated JS code in?
- `strict`: Be as strict as possible when checking for invalid code. This options enforces that all of your code is properly typed.
- `target`: Which JavaScript version should TSC compile your code to (ES3, ES5, ES2015, ES2016, etc.)?

- When input files are specified on the command line, tsconfig.json files are ignored.
  - For example: `tsc filename.ts`
  - use `tsc` instead

Type Annotations

We can specify the type using **`:type`** after the name of the variable, parameter or property.

TypeScript includes all the primitive types of JavaScript- number, string and boolean.

```
const grade: number = 90; // number variable
const name: string = "Anna"; // string variable
const isFun: boolean = true; // Boolean variable
```

Type Inference

It is not mandatory to annotate types in TypeScript, as it infers types of variables when there is no explicit information available in the form of type annotations.

```
let text = "some text";
text = 123; // Type '123' is not assignable to type 'string'
```

any

Never ever use the type **any** especially to silence an error or when you do not have prior knowledge about the type of some variables.

```
let something: any = 'Anna';
something = 569;
something = true;

let data: any[] = ["Anna", 569, true];
```

any is a type that disables type checking and effectively allows all types to be used.

enum

Enums allow us to declare **a set of named Constants**, a collection of related values that can be numeric or string values.

Enum values start from zero and increment by 1 for each member.

```
enum Technologies {
  Angular,
  React,
  ReactNative
}

// Technologies.React; returns 1
// Technologies["React"]; returns 1
// Technologies[0]; returns Angular
```

```
console.log(Technologies);

{
    '0': 'Angular',
    '1': 'React',
    '2', 'ReactNative',
    Angular: 0,
    React: 1,
    ReactNative: 2
}
```

Union Type

Union type allows us to combine more than one data type.

```
(type1 | type2 | type3 | .. | typeN)
```

```
let course: (string | number);
let data: string | number;
```

Explicit Array Type

There are two ways to declare an array type:

1. Using **square brackets**

```
let values: number[] = [12, 24, 48];
```

2. Using a **generic array type**, `Array<elementType>`

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

You can always initialize an array with many data types elements, but you will not get the advantage of TypeScript's type system.

Type alias

- A **type alias** in TypeScript is used to define custom types. It can define not only object shapes (similar to interfaces) but also primitive types, other types.

```
type Point = { x: number; y: number; };
const point: Point = { x: 10, y: 20 };

type ID = number | string;
let userID: ID = 123;
userID = "ABC";

type Greet = (name: string) => string;
const greet: Greet = (name) => `Hello, ${name}` ;
```

Type Assertion

1. Using the angular bracket <> syntax

```
let code: any = 123;
let courseCode = <number> code;
```

2. Using as keyword

```
let code: any = 123;
let courseCode = code as number;
```

It is purely a compile-time feature to assist with type-checking

Object Types

TypeScript has a specific syntax for typing objects.

```
const car: { type: string, model: string, year: number } = {
  type: "Toyota",
  model: "Corolla",
  year: 2009
};
```

Optional Property:

```
const car: { type: string, mileage?: number } = { // no error
  type: "Toyota"
};
car.mileage = 2000;
```

TypeScript can infer the types of properties based on their values.

```
const car = {
  type: "Toyota",
};

car.type = "Ford";
car.type = 2; // Error: Type 'number' is not assignable to type 'string'.
```

Object Types

We may represent object types as an **interface** or a **type alias**:

```
interface Person {
  name: string;
  age: number;
}
```

OR

```
type Person = {
  name: string;
  age: number;
};
```

```
function greet(person: Person) {
  return "Hello " + person.name;
}
```

Combine Types

```
type Person = {
    name: string;
    age: number;
};

type Tech = {
    phone: string;
    laptop: string;
};

type TechPerson = Person & Tech;

const anna: TechPerson = { name: "Anna", age: 20, phone: "Pixel", laptop: "MacBook Pro" };
```

Combine Interfaces

```
interface Person {
  name: string;
  age: number;
}

interface Tech {
  phone: string;
  laptop: string;
}

const anna: Person & Tech = { name: "Anna", age: 0, phone: "Pixel", laptop: "MacBook Pro" };
```

Utility Types

Utility types help transform and manipulate types to make your code more flexible and reusable. These utility types can extract, combine, or modify types in various ways.

```
interface Person {
   firstname: string;
   lastname: string;
   age: number;
   height: number;
   weight: number;
}

type NamedPerson = Pick<Person, "firstname" | "lastname">;
const anna: NamedPerson = { firstname: "Anna", lastname: "Smith" };

type SomePerson = Partial<Person>;
const mike: SomePerson = { firstname: "Mike", height: 180 };

type JustName = Omit<Person, "age" | "height" | "weight">;
const smith: JustName = { firstname: "Smith", lastname: "John" };
```

Interface

Interface is a structure that defines the **contract** in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

An interface is defined with the keyword `interface` and it can include properties and method declarations using a function or an arrow function.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given. **Each parameter in the parameter list requires both name and type.**

```
interface IEmployee {
    empCode: number;
    empName: string;
    setEmpName(name: string): void;
    getEmpName: () => string;
}
```

```
let emp: IEmployee = {
    empCode: 1001,
    empName: 'John',
    setEmpName: function (name: string): void {
        this.empName = name;
    },
    getEmpName: function () {
        return this.empName;
    }
}
emp.setEmpName('Edward');
console.log(emp.getEmpName());
```

Interface as Type

Interface in TypeScript can be used to **define a type** and also to **implement** it in the class. We can have optional properties, marked with a "?". We can mark a property as read only.

```
interface IKeyPair {
    readonly key: number;
    value?: string;
}

let kv1: IKeyPair = { key: 1, value: "John" };
let kv2: IKeyPair = { key: 2 };
let kv3: IKeyPair = { key: 2, age: 20 }; // Compiler error
kv2.key = 3; // Compiler error
```

Extending Interfaces

Interfaces can extend one or more interfaces. The object from the extended interface **must include all the properties and methods from both interfaces**, otherwise, the compiler will show an error.

```
interface ICity {
    name: string;
}

interface IAddress extends ICity {
    zipcode: number;
}

let northStreet: IAddress = {
    zipcode: 52557,
    name: "Fairfield"
}
```

Implementing an interface

Interfaces can be implemented with a Class. The Class implementing the interface needs to **strictly conform to the structure of the interface**.

The implementing class can define extra properties and methods, but at least it must define all the members of an interface. A class can implements multiple interfaces.

```
interface ICourse {
    code: number;
    name: string;
    grade: number;
    setGrade(grade: number): void;
    getGrade(): number;
}
```

```
class Course implements ICourse {
    code: number;
    name: string;
    grade: number = 0;

    constructor(code: number, name: string) {
        this.code = code;
        this.name = name;
    }
    setGrade(grade: number): void {
        this.grade = grade;
    }
    getGrade(): number {
        return this.grade;
    }
}

let course = new Course(472, "Web Application Programming");
```

Class

Classes are the fundamental entities used to create reusable objects. Functionalities are passed down to other classes and objects can be created from classes.
The class in TypeScript is compiled to plain JavaScript function constructor or class by the TS compiler to work across platforms and browsers.
A class can include the following:

- Constructor
  - The constructor is a special method which is called when creating an object. An object of the class can be created using the **`new`** keyword.

    If there's no constructor being defined manually, a default one (depends on situation) will be used to create objects.
  - Only 1 constructor is allowed in a class.
- Properties
- Methods

Inheritance

- TypeScript classes can be extended to create new classes with inheritance, using the `extends` keyword.

```
class B extends A {}
```

- This means that the `B` class now includes all the members of the A class.
- The constructor of the `B` class initializes its own members as well as the parent class's properties using the `super` keyword.
- **Classes can only extend a single class.**
- **A class can implements multiple interfaces.**
- **Constructors for derived classes must contain a 'super' call.**
- **In subclass, if there's no constructor provided, it'll use super class's one. 'super' must be called before accessing 'this' in the constructor of a derived class.**

Inheritance Example

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

class MSD extends Course {
    code: number;
    constructor(code: number, name: string) {
        super(name);
        this.code = code;
    }
    displayName(): void {
        console.log(`Name = ${this.name}, Course Code = CS${this.code}`);
    }
}

let course = new MSD(472, "Web Application Programming");
course.displayName();
```

A class can implement multiple interfaces

```
interface ICourse {
   name: string;
   display(): void;
}

interface ICode {
   code: number;
}

class MAP implements ICourse, ICode {
   code: number;
   name: string;

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

   display(): void {
      console.log(`${this.name}, Course Code = CS${this.code}`);
   }
}

let wap: MAP = new MAP(472, "Web Application Programming");
wap.display();
```

- The MAP class implements two interfaces - **ICourse** and **ICode**. So, an instance of the MAP class can be assigned to a variable of **ICourse** or **ICode** type. However, an object of type **ICode** cannot call the **display()** method because **ICode** does not include it.

Method Overriding

```
class Meditator {
    name: string;
    constructor(name: string) { this.name = name }
    meditate(duration: number = 20) {
        console.log(this.name + " is meditating for " + duration + " mins!");
    }
}

class Sidha extends Meditator {
    constructor(name: string) { super(name) }
    meditate(duration: number = 40) {
        console.log('Meditation started')
        super.meditate(duration);
    }
}

let john = new Sidha("John");
john.meditate();  // Meditation started John is meditating for 40 mins!
```

- When a child class defines its own implementation of a method from the parent class, it is called method overriding.

Abstract Class

Define an abstract class in Typescript using the `abstract` keyword.

Abstract classes are mainly for inheritance where other classes may derive from them. **We cannot create an instance of an abstract class**.

An abstract class includes one or more **`abstract` methods or properties**.

The class which extends the abstract class **must** implement all the abstract methods and properties.

An **`abstract`** class doesn't need to have abstract methods or properties. If a class has **`abstract`** method or properties, must declare as **`abstract`**.

Mostly used when child classes want to share the some but not all behavior, it should be used primarily for objects that are closely related.

Abstract Class Example

```
abstract class Employee {
    fname: string;
    lname: string;
    salary: number;
    abstract address: string;

    constructor(fname: string, lname: string, salary: number){
        this.fname = fname;
        this.lname = lname;
        this.salary = salary;
    }

    abstract computeAnnualSalary(): number;
}
```

```
class HourlyEmployee extends Employee {
    address: string = 'default';
    hoursPerWeek: number;

    constructor(fname: string, lname: string, salary: number, hour
sPerWeek: number) {
        super(fname, lname, salary);
        this.hoursPerWeek = hoursPerWeek;
    }

    computeAnnualSalary(): number {
        return this.salary * this.hoursPerWeek * 52;
    }
}

let john = new HourlyEmployee('John', 'Smith', 30, 40);
console.log(john.computeAnnualSalary());
console.log(john.address);
```

Access Modifiers

There are three types of access modifiers: **`public`**, **`private`** and **`protected`**. Encapsulation is used to control class members' visibility.

public

- By default, all members of a class in TypeScript are `public`. All the public members can be accessed anywhere without any restrictions.

```
class Course {
   public code: string = "472";
   name: string = "Node.js";
}

let course = new Course();
course.code = "CS445";
course.name = "MAP";
```

- **code**and **name** are accessible outside of the class using an object of the class.

Class Example - Shortcut

Adding **access modifiers** to the constructor arguments lets the class know that they're properties of a class. If the arguments don't have access modifiers, they'll be treated as an argument for the constructor function and not properties of the class.

```
interface Book {
    bookName: string;
    isbn: number;
}

class Course {

    // public is shorthand for this.name = name, this.code = code
    constructor(public name: string, public code: number) { }

    useBook(book: Book) {
        console.log(`Course ${this.name} is using the textbook:
                ${book.bookName} who's ISBN = ${book.isbn}`);
    }
}
```

private

The `private` access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.

```
class Course {
    private code: string = "CS477";
    name: string= "Node.js";
}

let course = new Course();
course.code = "CS445"; // Compiler Error
console.log(course.code); // Compiler Error
course.name = "MAP"; // OK
```

protected

The `protected` access modifier is similar to the `private` access modifier, except that protected members can be accessed using their deriving classes.

```
class Course {
    public name: string;
    protected code: number;
    constructor(name: string, code: number) {
        this.name = name;
        this.code = code;
    }
}

class MAPCourse extends Course {
    private details: string;
    constructor(name: string, code: number, department: string) {
        super(name, code);
        this.details = `${department} - ${this.code}`;
    }
}

let map = new MAPCourse("Web Application Programming", 472, "Computer Science");
map.code; // Compiler Error
```

Property **code** is protected and only accessible within class **Course** and its subclasses.

Readonly

Read-only members can be accessed outside the class, but their value cannot be changed. Since read-only members cannot be changed outside the class, they either need to be initialized at declaration or initialized inside the class constructor.

```
class Course {
    readonly code: number;
    name: string;
    constructor(code: number, name: string) {
        this.code = code;
        this.name = name;
    }
}
let course = new Course(569, "WAD");
course.code = 445; // Compiler Error
course.name = 'Modern Asynchronous Programming'; // Ok
```

static

- ES6 includes static members and so does TypeScript.
- The static members of a class are accessed using the class name and dot notation, without creating an object.

```
class Circle {
   static pi: number = 3.14;


   static calculateArea(radius: number) {
       return this.pi * radius * radius;
   }
}
Circle.pi; // returns 3.14
Circle.calculateArea(5); // returns 78.5
```