TypeScript

Classes

Content

1. Class
2. Access Modifiers
3. The readonly Modifier
4. Getters and Setters
5. Inheritance
6. Static Methods & Properties
7. Abstract Classes

1. 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}`;
    }
}
```

2. Access Modifiers

- TypeScript provides three access modifiers:
  - private: allows access within the same class
  - protected: allows access within the same class and subclasses
  - public: allows access from any location (default)

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

3. The readonly Modifier

- The readonly modifier that allows you to mark the properties of a class immutable

```
class Person {
    readonly birthDate: Date;

    constructor(birthDate: Date) {
        this.birthDate = birthDate;
    }
}
```

```
let person = new Person(new Date(1990, 12, 25));
person.birthDate = new Date(1991, 12, 25); // Compile error
```

4. Getters and Setters

- A getter method returns the value of the property's value. A getter is also called an accessor.
- A setter method updates the property's value. A setter is also known as a mutator.

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

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

```
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 getFullName(): string {
        return `${this._firstName} ${this._lastName}`;
    }
}
```

5. Inheritance

- Parent Class

```
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}.`;
    }
}
```

- Constructor

```
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());
```

- **Method overriding**

```
class Employee extends Person {
    constructor(
        firstName: string,
        lastName: string,
        private jobTitle: string) {

        super(firstName, lastName);
    }

    describe(): string {
        return super.describe() + `I'm a ${this.jobTitle}.`;
    }
}
```

6. Static Methods & Properties

- Static properties

```
class Employee {
    static headcount: number = 0;
    constructor(
        private firstName: string,
        private lastName: string,
        private jobTitle: string) {
        Employee.headcount++;
    }
}
```

```
let john = new Employee('John', 'Doe', 'Front-end Developer');
let jane = new Employee('Jane', 'Doe', 'Back-end Developer');
console.log(Employee.headcount); // 2
```

- **Static methods**

```
class Employee {
    private static headcount: number = 0;
    constructor(
        private firstName: string,
        private lastName: string,
        private jobTitle: string) {
        Employee.headcount++;
    }
    public static getHeadcount() {
        return Employee.headcount;
    }
}
```

```
let john = new Employee('John', 'Doe', 'Front-end Developer');
let jane = new Employee('Jane', 'Doe', 'Back-end Developer');
console.log(Employee.getHeadcount); // 2
```

7. Abstract Classes

- An abstract class is typically used to define common behaviors for derived classes to extend. An abstract class cannot be instantiated directly.

```
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.`;
    }
}
```

```
class FullTimeEmployee extends Employee {
    constructor(firstName: string, lastName: string, private salary: number) {
        super(firstName, lastName);
    }
    getSalary(): number {
        return this.salary;
    }
}
```

```
class Contractor extends Employee {
    constructor(firstName: string, lastName: string,
                private rate: number, private hours: number) {
        super(firstName, lastName);
    }
    getSalary(): number {
        return this.rate * this.hours;
    }
}
```

```
let john = new FullTimeEmployee('John', 'Doe', 12000);
let jane = new Contractor('Jane', 'Doe', 100, 160);

console.log(john.compensationStatement());
console.log(jane.compensationStatement());
```

- Output

```
John Doe makes 12000 a month.
Jane Doe makes 16000 a month.
```

THE END