# Classes

## Prototypal inheritance

JS supports something called prototypal inheritance

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype.toString = function () {
    return this.x + ", " + this.y;
};
function ColoredPoint(x, y, color) {
   Point.call(this, x, y);
   this.color = color
}
ColoredPoint.prototype = Object.create(Point.prototype);
ColoredPoint.prototype.toString = function () {
    return Point.prototype.toString.call(this) + ' in ' + this.color;
}
const coloredPoint = new ColoredPoint(1, 2, 'red');
coloredPoint.toString(); // => 1, 2 in red
```

This isn't `class`y.

## Classes

Since ES2015 we can use classes

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() { return `${this.x}, ${this.y}`; }
}
class ColoredPoint extends Point {
    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }
    toString() {
        return `${super.toString()} in ${this.color}`;
    }
}
```

## JS classes

JS classes are a big improvement!

- Looks like classes in C# and Java
- Required to use `new` to call the constructor
- Inherit with `extends`
- Support for `static` members
- Support for properties (with `get` and `set`)
- Support for *private* with `#` (new in JavaScript)

But still missing some features.

- No `abstract` classes
- No is `protected`
- Dynamically typed (of course)

## Classes in TypeScript

Classes in TypeScript look slightly different

```
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x; this.y = y;
    }
    toString() { return `${this.x}, ${this.y}`; }
}

class ColoredPoint extends Point {
    private color: string;
    constructor(x: number, y: number, color: string) {
        super(x, y); this.color = color;
    }
    toString() { return `${super.toString()} in ${this.color}`; }
}
```

---

## Classes in TypeScript

Classes in TypeScript are more complete.

- Supports `abstract` classes and methods
- Supports `private`, `public`, `protected` and `#` (ES private)
- Supports generic classes
- Supports explicit interface implementation with `implements`
- A class is always associated with a static type

```
const point: ColoredPoint = new ColoredPoint(1, -3, 'red');
```

---

### Some TypeScript examples

```
abstract class Animal {
    protected abstract kind: string;
}
new Animal(); // Error: Cannot create an instance of an abstract class.

interface Named {
    name: string;
}

class Dog implements Named {
    // Error: Class 'Dog' incorrectly implements interface 'Named'
    protected kind = 'Dog';
}

new Dog().kind;
// Error: Property 'kind' is protected and only accessible within class
'Dog' and its subclasses.
```

## Constructor

A class can have exactly one `constructor` implementation.

```
class Point {
    private x: number;
    private y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}
```

Constructors support the shortened syntax

```
class Point {
    constructor(private x: number, private y: number) { }
}
```

## Inheritance

A common object-oriented pattern is *inheritance*.

```
class Point {
    constructor(protected x: number, protected y: number) { }
    distance(other: Point) {
        return Math.sqrt(Math.pow(this.x - other.x, 2) +
            Math.pow(this.y - other.y, 2));
    }
}
```

```
class Point3D extends Point {
    constructor(x: number, y: number, protected z: number) {
        super(x, y);
    }
    distance(other: Point) {
        const distance2D = super.distance(other);
        if (other instanceof Point3D) {
            return Math.sqrt(Math.pow(distance2D, 2)
                + Math.pow(this.z - other.z, 2));
        } else {
            return distance2D;
        }
    }
}
```

---

## Access modifiers

TypeScript supports `private`, `protected`, `public`

```
class Point3D {
    constructor(private x: number, protected y: number, public z: number)
{ }
}

class Child extends Point3D {
    constructor(x: number, y: number, z: number) { super(x, y, z); }

    public getX() { return this.x; }
    // => Error: Property 'x' is private and only
    // accessible within class 'Point3D'
}

new Point3D(0, 0, 0).y;
// => Error Property 'y' is protected and only
// accessible within class 'Point3D' and its subclasses.
```

**Question:** How does this translate to JavaScript?

---

## Accessors

TypeScript supports ES6's `get` and `set`.

```
class Person {
    #firstName: string;
    #lastName: string;

    get fullName() {
        return `${this.#firstName} ${this.#lastName}`;
    }

    set firstName(value: string) {
        this.#firstName = value;
    }
    set lastName(value: string) {
        this.#lastName = value;
    }
}
```

```
const p = new Person();
p.firstName = 'Albert';
p.lastName = 'Einstein';
p.fullName; // => Albert Einstein
```

---

## Static properties

Use the `static` keyword to create properties belonging to the class itself, not instances of the class.

```
class Point {
    constructor(public x: number, public y: number) {
        Point.origin; // => OK
        this.origin;
        // => error! Property 'origin' does not exist on type 'Point'.
    }

    static origin = new Point(0, 0);
    static parse(pointValues: { x: number, y: number }) {
        return new Point(pointValues.x, pointValues.y);
    }
}
Point.parse({ x: 3, y: -5});
```

---

### Readonly modifier

You can use the `readonly` keyword to make a property readonly and force initialization in constructor or declaration.

```
class Person {
    static readonly favoriteDrink: 'beer';

    constructor(readonly birthDate: Date) { }
}

const p = new Person(new Date(1986, 4, 30));
p.birthDate = new Date(1993, 2, 4);
// => error! Cannot assign to 'birthDate' because it is a
//  constant or a read-only property.

Person.favoriteDrink = 'water';
// => error! Cannot assign to 'favoriteDrink' because it is a
//  constant or a read-only property.
```

---

### Strict property initialization

You can use `--strictPropertyInitialization`
(in combination with `--strictNullChecks`)

```
class Person {
    name: string;

    constructor() { }
}
// => error: Property 'name' has no initializer and is not definitely
assigned in the constructor.
```

All properties need to be initialized in the constructor.

```
class Person {
    name: string;

    constructor() {
        this.name = 'foobar';
     }
}
// => OK
```

---

#### Definite assignment assertion

Let us say we initialize a field outside of the constructor.

```
class Person {
    name: string;

    constructor() { this.initializeName(); }

    initializeName() { this.name = ''; }
}
// => error: Property 'name' has no initializer and is not definitely
assigned in the constructor.
```

To enable this, use the "definite assignment assertion" (`!`).

```
class Person {
    name!: string;
    //  ^ indicate that the name will be assigned

    constructor() { this.initializeName(); }

    initializeName() { this.name = ''; }
}
```

## Type association

A class can be used as an interface.

```
class Point {
    constructor(public x: number, public y: number){}
}

interface Coordinated extends Point {
    z: number;
}

class OriginPoint implements Point {
    x = 0;
    y = 0;
}
```

**Question:** What happens here?

## Constructor functions

`class MyClass {}` does 2 things.

1. Create the *class* in JavaScript (called `MyClass`)
2. Declares the *interface* which describes the *shape* for *instances* of `MyClass`.

```
class Point {
    constructor(public x: number, public y: number){ }
}
const PointCopy = Point;
```

**Question:** What is the type of the PointCopy here?

```
interface PointConstructor {
    new(x: number, y: number): Point;
}
```

## ES Private

There is a new kind of private with `#`

```
class Point {
    #x: number; #y: number;
    constructor(x: number, y: number) {
        this.#x = x; this.#y = y;
    }

    equals(other: Point) {
        return other.#x === this.#x && this.#y === other.#y;
    }
}
const p1 = new Point(0, 1);
const p2 = new Point(0, 1);

p1.equals(p2); // => true
p1.#x;         // => Error: Property '#x' is not accessible outside class 'Point'
```

ES private feature *hard privacy*, it cannot be accessed from the outside!

---

## Private vs ES Private in TypeScript

| | **TypeScript `private`** | **ES `#private`** |
|---|---|---|
| **Lowest supported target** | `ES3` | `ES2015` |
| **Implementation** | Erased | `WeakMap` * |
| **Runtime privacy** | Soft | Hard |
| **Performance overhead** | None | Small ** |
| **Property parameter syntax** | Yes | No |

---

- * Unless target is `ESNext`
- ** Guesstimate, depends on the JS engine

---

## Method overloading

Method overloading can be used with classes.

```
interface SuccessHandler {
    (data: any, textStatus: string): void;
}

class JQuery {
    get(url: string): any;
    get(url: string, handler?: SuccessHandler) {
        // do stuff
    }
}
```

You can even overload a `constructor`.

**Question:** What happens if your remove the `?` from `handler`?

---

### Method overloading limitations

- Overloaded methods share one implementation
- All overloads must be compatible with the implementation

**Rule of thumb** only use method overloading if it actually adds functionality

```
// Don't do this!
function findPerson(id: number): Person
function findPerson(id: string): Person
function findPerson(id: number | string): Person {
  /* .. */
}
```

---