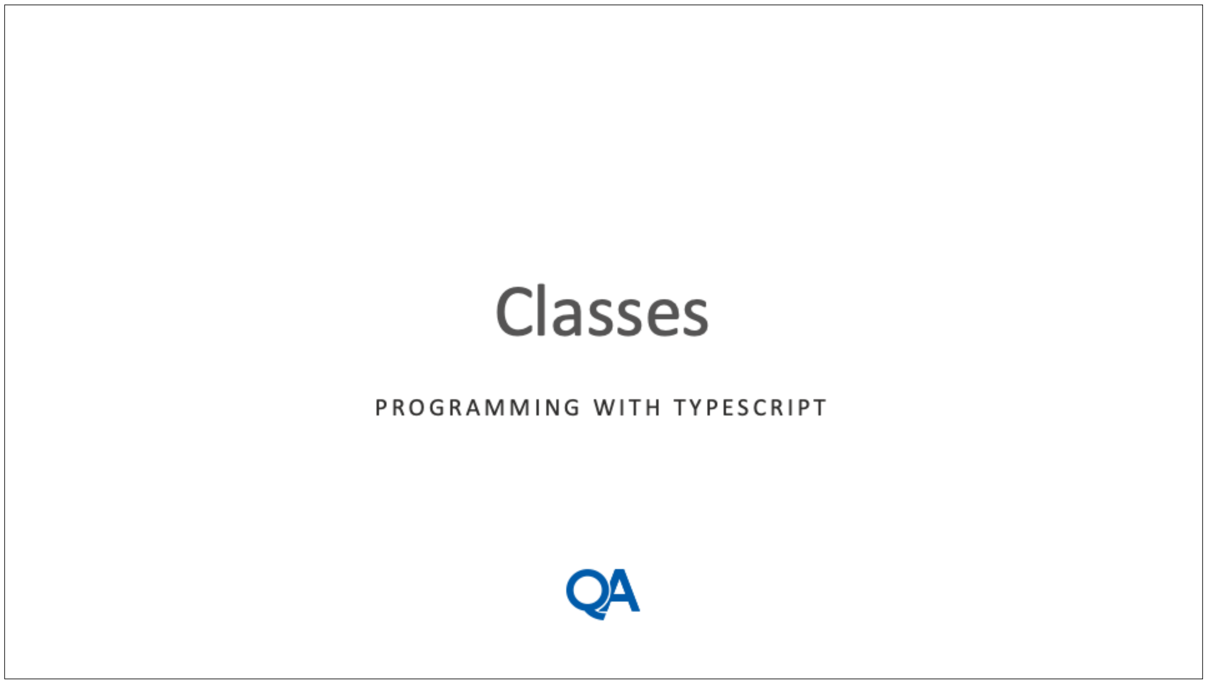
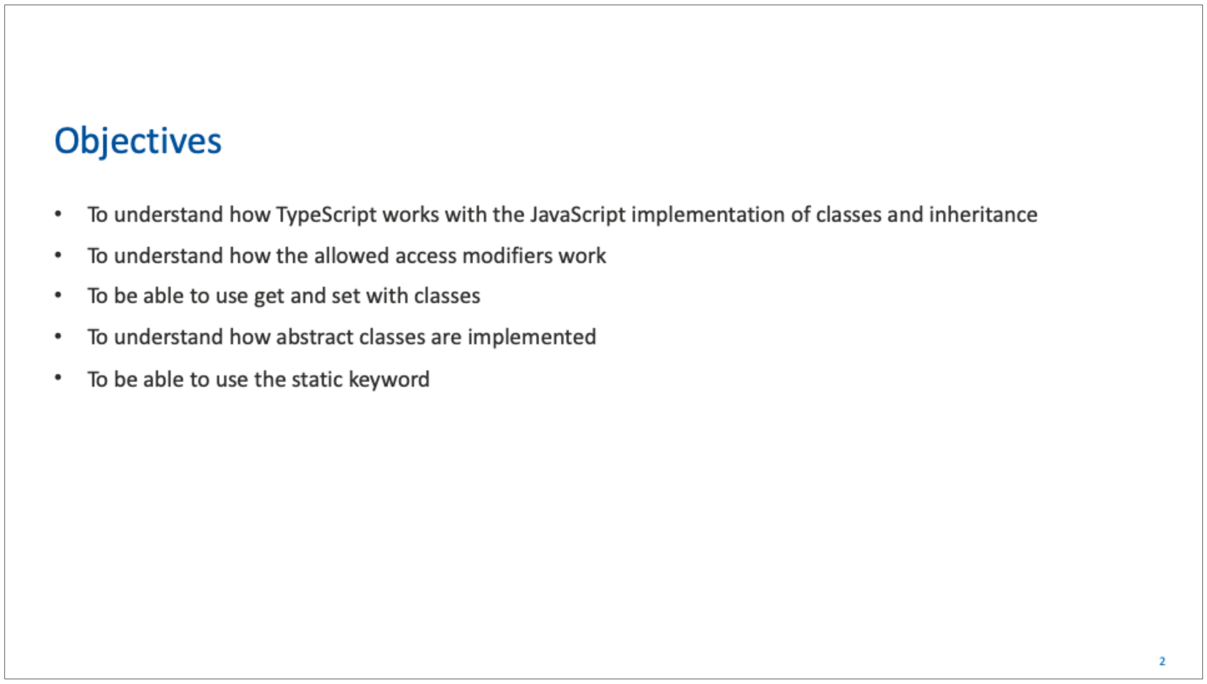


# Classes

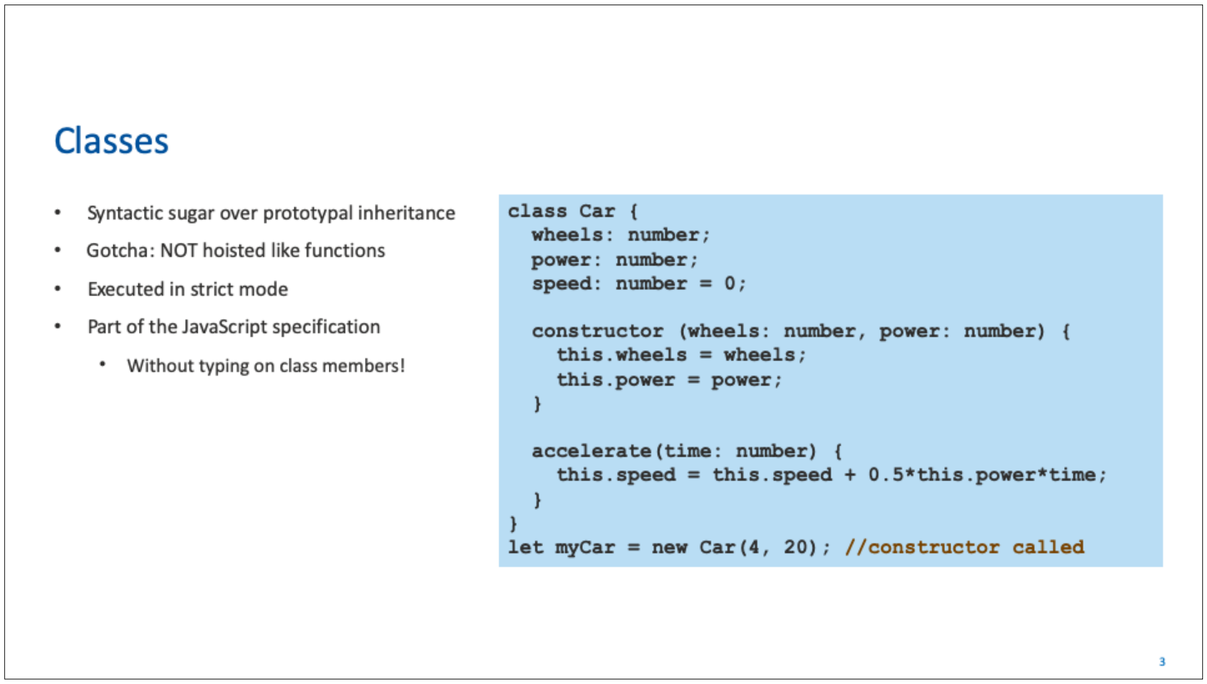
PROGRAMMING WITH TYPESCRIPT



## Objectives

- To understand how TypeScript works with the JavaScript implementation of classes and inheritance
- To understand how the allowed access modifiers work
- To be able to use get and set with classes
- To understand how abstract classes are implemented
- To be able to use the static keyword

2



## Classes

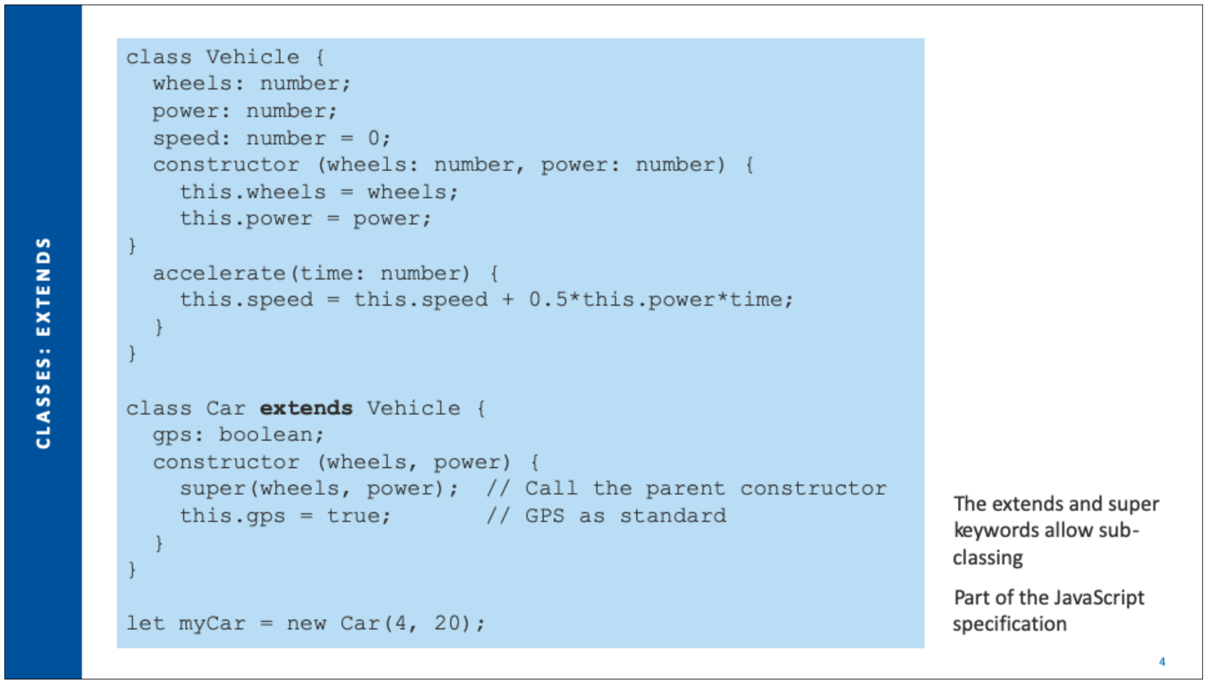
- Syntactic sugar over prototypal inheritance
- Gotcha: NOT hoisted like functions
- Executed in strict mode
- Part of the JavaScript specification
  - Without typing on class members!

```
class Car {
  wheels: number;
  power: number;
  speed: number = 0;

  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }

  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
let myCar = new Car(4, 20); //constructor called
```

3

Constructor method is called when the class is instantiated through the “new” keyword

Methods can be created without using the function keyword or assigning to “this”



CLASSES: EXTENDS

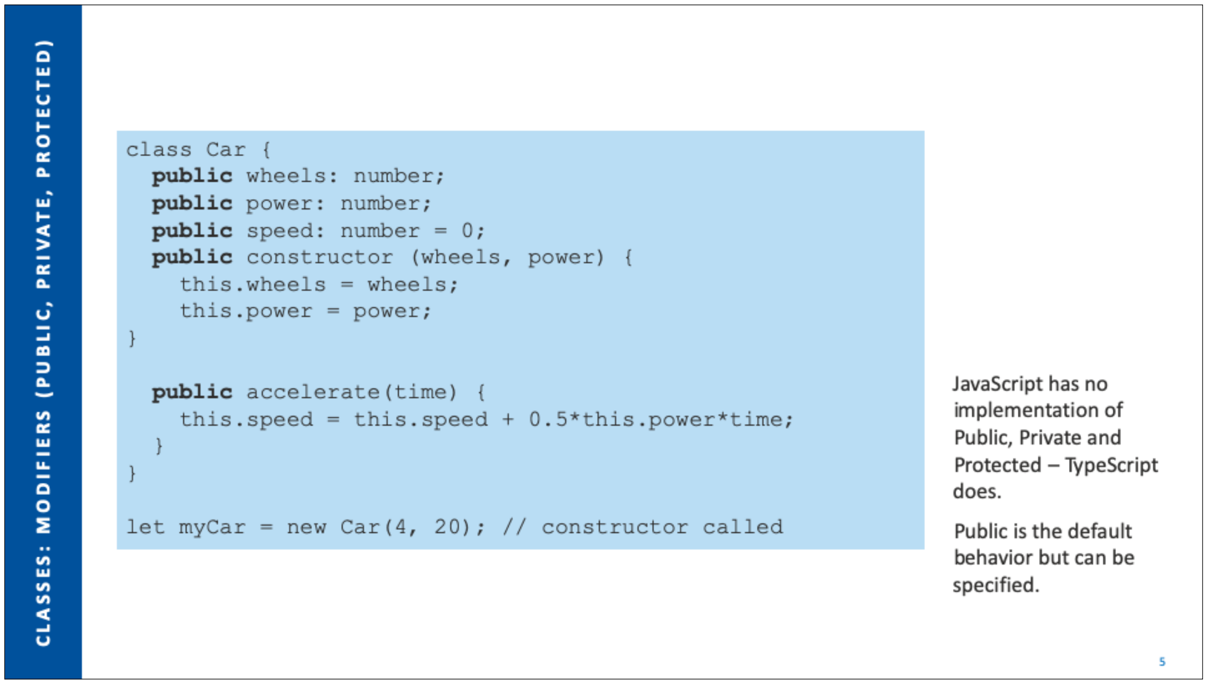
```
class Vehicle {
  wheels: number;
  power: number;
  speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
}
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) {
    super(wheels, power);  // Call the parent constructor
    this.gps = true;       // GPS as standard
  }
}

let myCar = new Car(4, 20);
```

The extends and super keywords allow sub-classing

Part of the JavaScript specification

4

GOTCHA! : “this” will be undefined before you call “super()” in a subclass



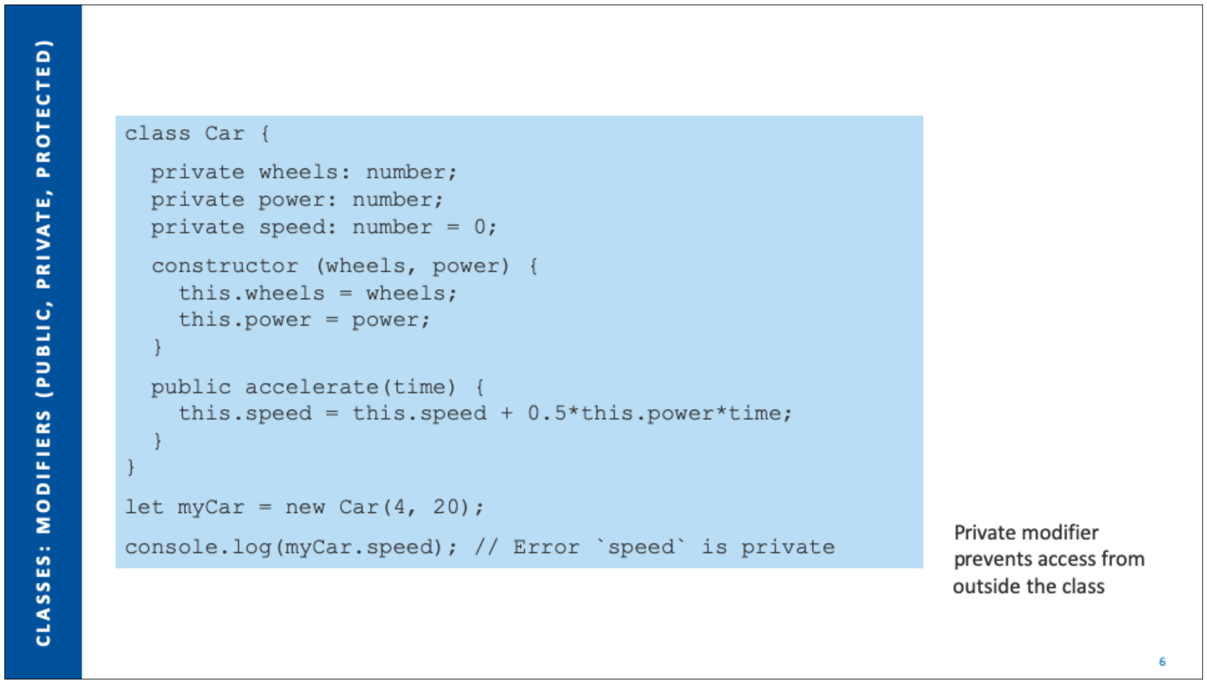
## CLASSES: MODIFIERS (PUBLIC, PRIVATE, PROTECTED)

```
class Car {
  public wheels: number;
  public power: number;
  public speed: number = 0;
  public constructor (wheels, power) {
    this.wheels = wheels;
    this.power = power;
}

  public accelerate(time) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

let myCar = new Car(4, 20); // constructor called
```

JavaScript has no implementation of Public, Private and Protected – TypeScript does.

Public is the default behavior but can be specified.

5

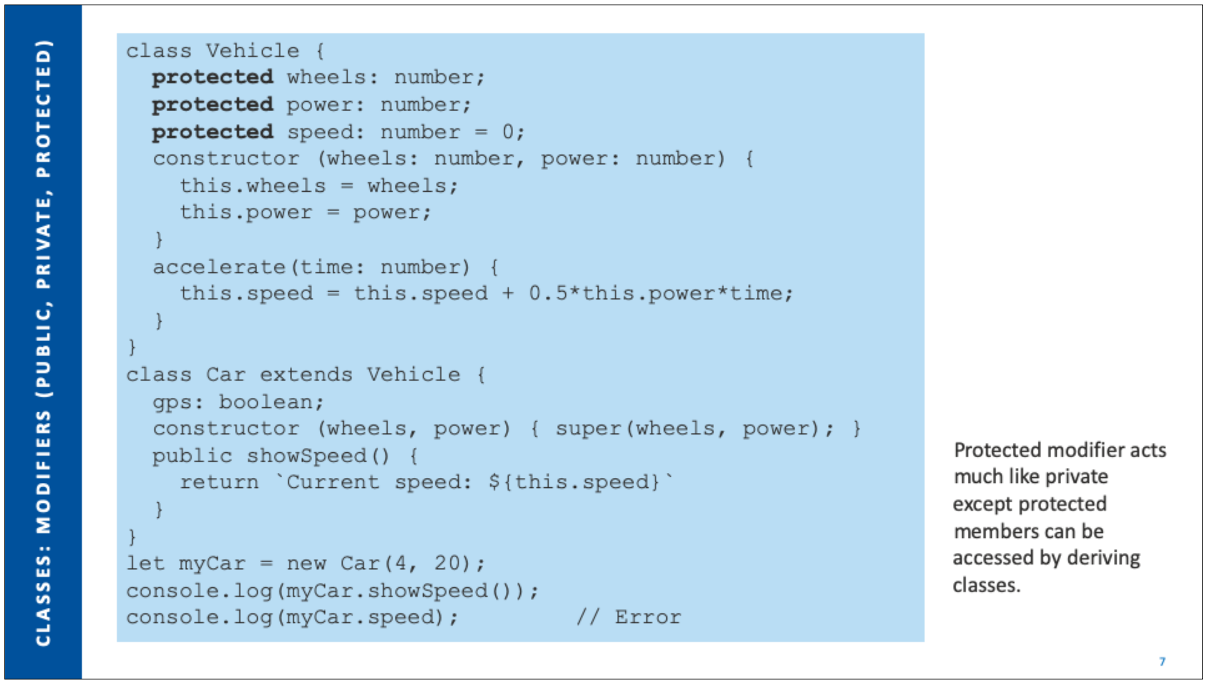


## CLASSES: MODIFIERS (PUBLIC, PRIVATE, PROTECTED)

```
class Car {
  private wheels: number;
  private power: number;
  private speed: number = 0;
  constructor (wheels, power) {
    this.wheels = wheels;
    this.power = power;
  }
  public accelerate(time) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
let myCar = new Car(4, 20);
console.log(myCar.speed); // Error `speed` is private
```

Private modifier prevents access from outside the class

6

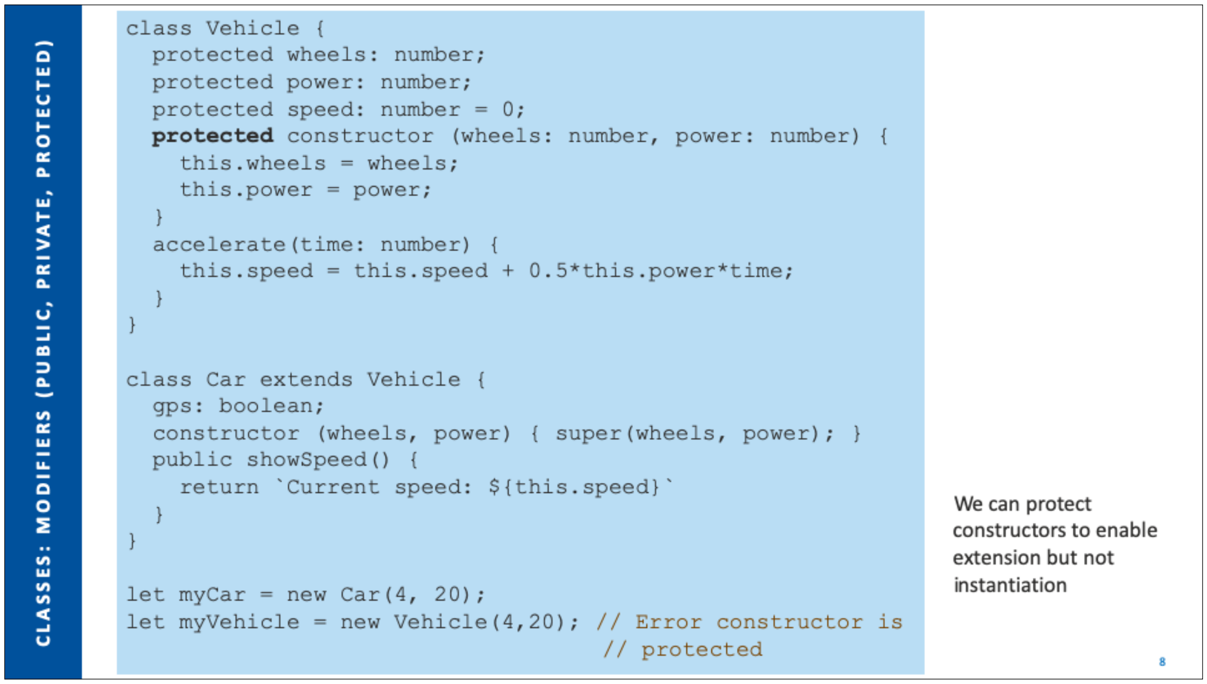


## CLASSES: MODIFIERS (PUBLIC, PRIVATE, PROTECTED)

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  protected speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) { super(wheels, power); }
  public showSpeed() {
    return `Current speed: ${this.speed}`
  }
}
let myCar = new Car(4, 20);
console.log(myCar.showSpeed());
console.log(myCar.speed);          // Error
```

Protected modifier acts much like private except protected members can be accessed by deriving classes.

7

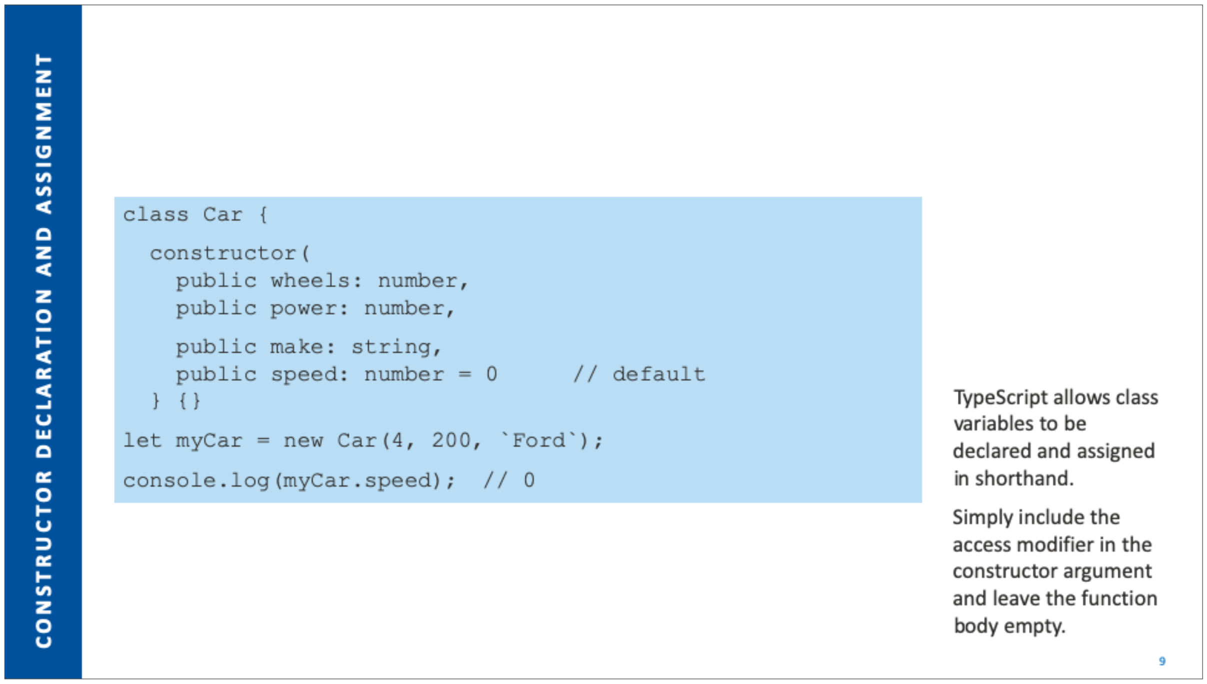


## CLASSES: MODIFIERS (PUBLIC, PRIVATE, PROTECTED)

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  protected speed: number = 0;
  protected constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  gps: boolean;
  constructor (wheels, power) { super(wheels, power); }
  public showSpeed() {
    return `Current speed: ${this.speed}`
  }
}

let myCar = new Car(4, 20);
let myVehicle = new Vehicle(4,20); // Error constructor is
                                    // protected
```

We can protect constructors to enable extension but not instantiation

8



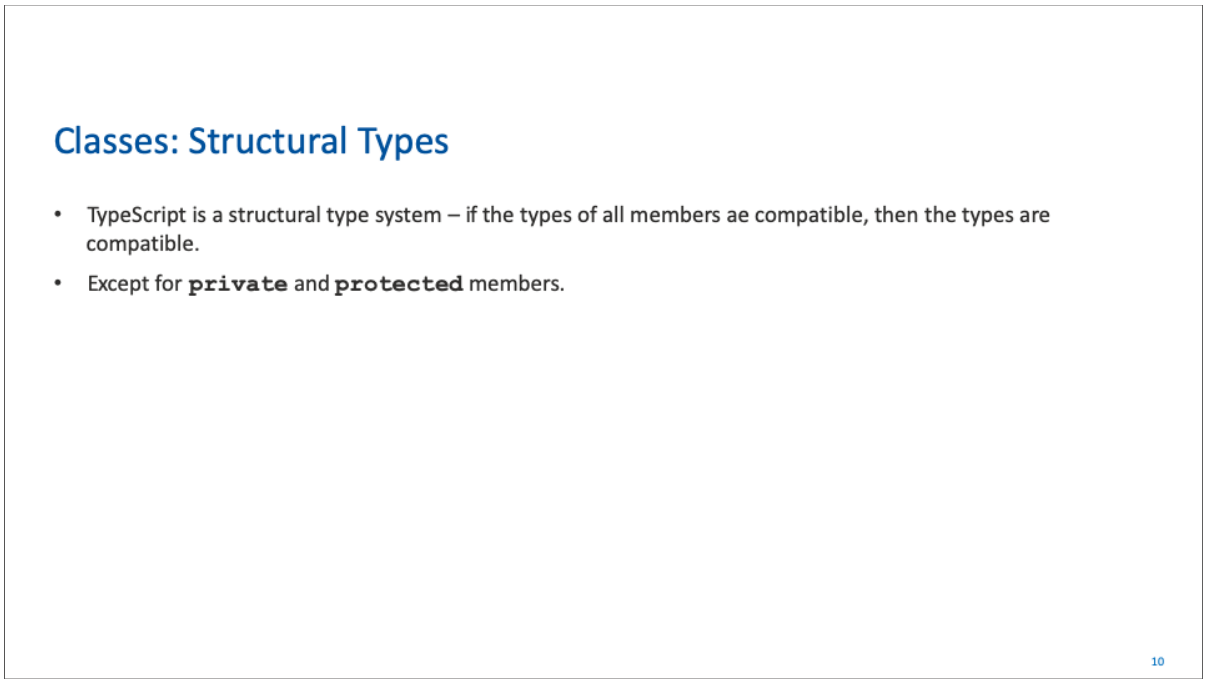
## CONSTRUCTOR DECLARATION AND ASSIGNMENT

```
class Car {
  constructor(
    public wheels: number,
    public power: number,
    public make: string,
    public speed: number = 0      // default
  ) {}
let myCar = new Car(4, 200, `Ford`);
console.log(myCar.speed);  // 0
```

TypeScript allows class variables to be declared and assigned in shorthand.

Simply include the access modifier in the constructor argument and leave the function body empty.

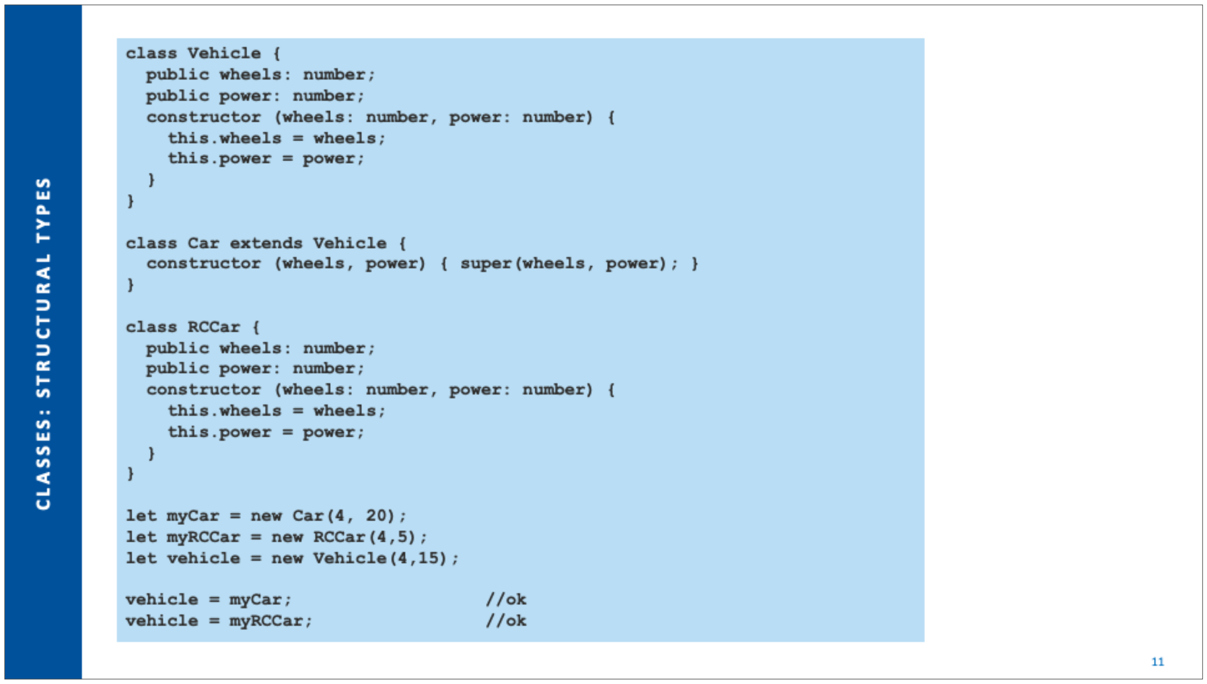
9



## Classes: Structural Types

- TypeScript is a structural type system – if the types of all members ae compatible, then the types are compatible.
- Except for **`private`** and **`protected`** members.

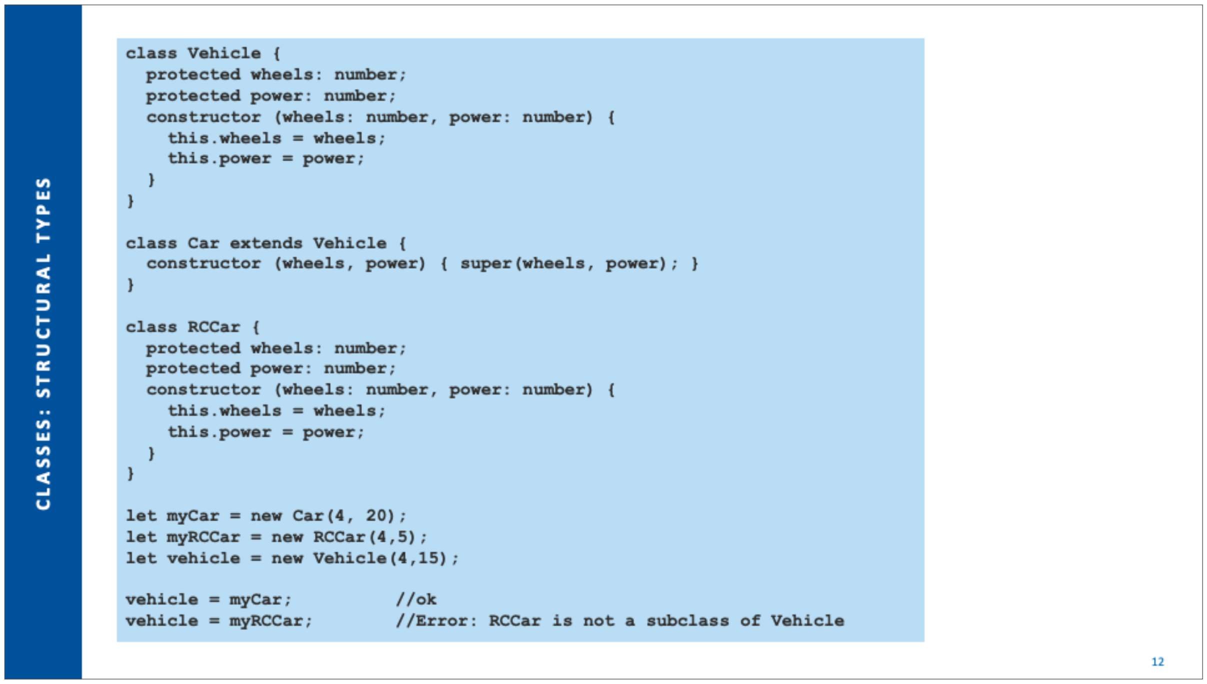
10



## CLASSES: STRUCTURAL TYPES

```
class Vehicle {
  public wheels: number;
  public power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
}

class RCCar {
  public wheels: number;
  public power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

let myCar = new Car(4, 20);
let myRCCar = new RCCar(4,5);
let vehicle = new Vehicle(4,15);

vehicle = myCar;                  //ok
vehicle = myRCCar;                //ok
```

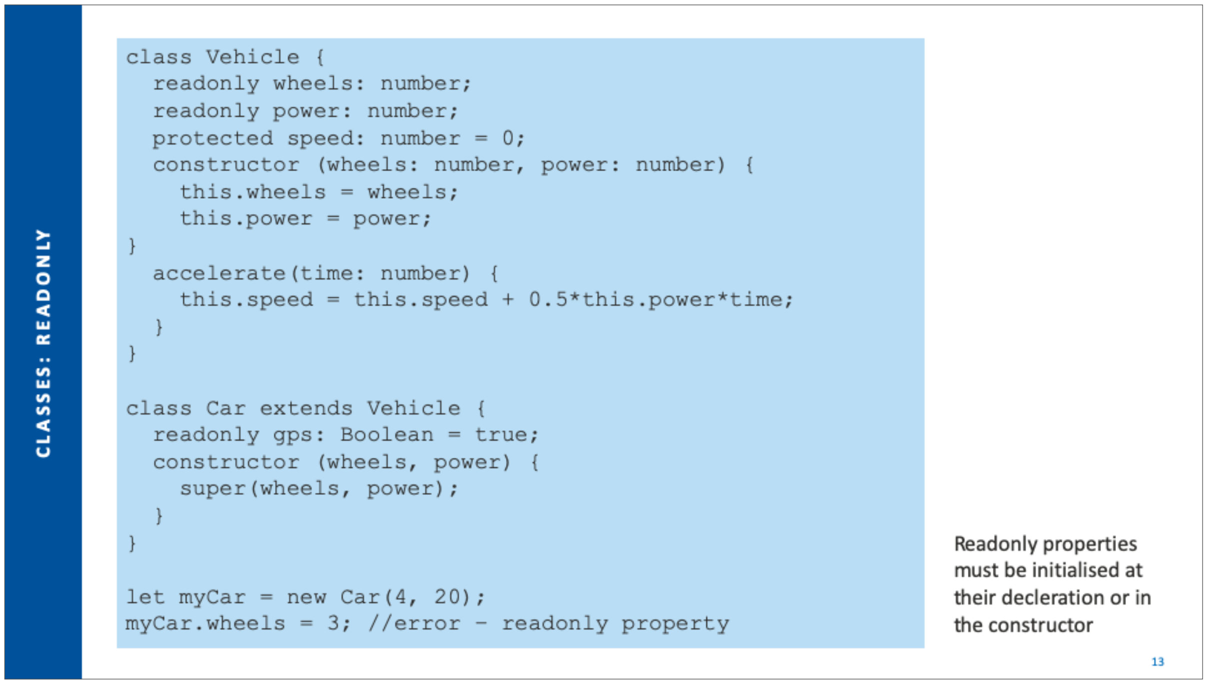
11



## CLASSES: STRUCTURAL TYPES

```
class Vehicle {
  protected wheels: number;
  protected power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
}

class RCCar {
  protected wheels: number;
  protected power: number;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
}

let myCar = new Car(4, 20);
let myRCCar = new RCCar(4,5);
let vehicle = new Vehicle(4,15);

vehicle = myCar;          //ok
vehicle = myRCCar;        //Error: RCCar is not a subclass of Vehicle
```

12

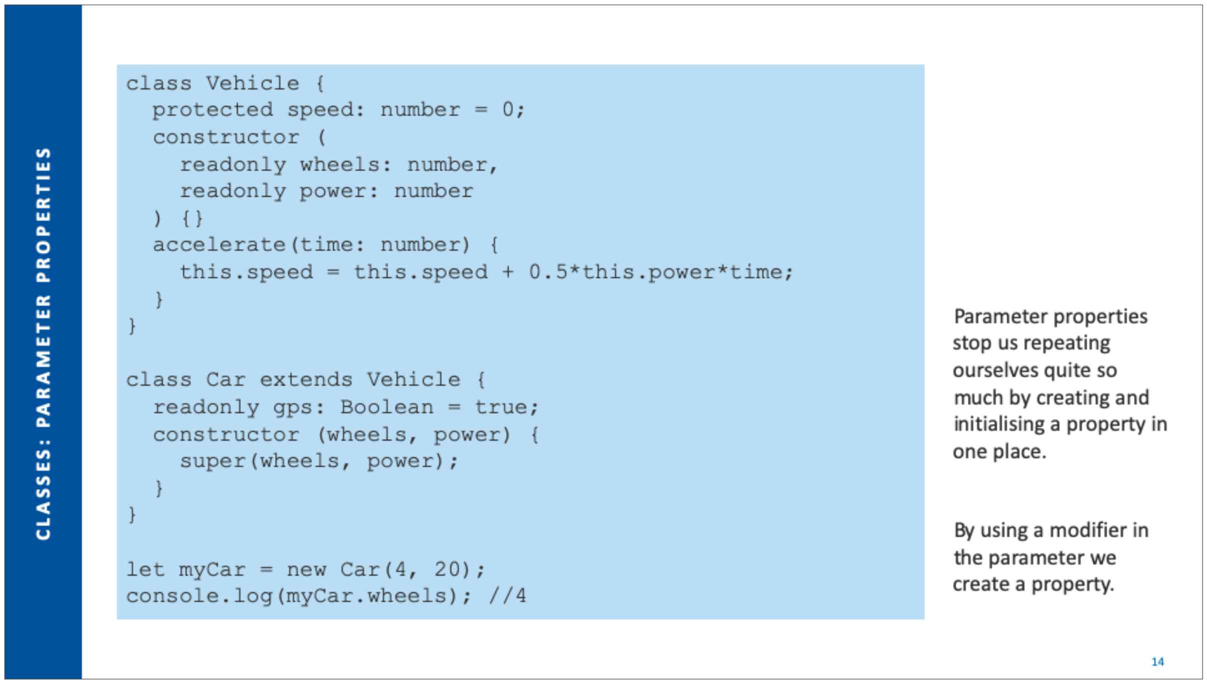


## CLASSES: READONLY

```
class Vehicle {
  readonly wheels: number;
  readonly power: number;
  protected speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
}
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  readonly gps: Boolean = true;
  constructor (wheels, power) {
    super(wheels, power);
  }
}

let myCar = new Car(4, 20);
myCar.wheels = 3; //error - readonly property
```

Readonly properties must be initialised at their decleration or in the constructor

13



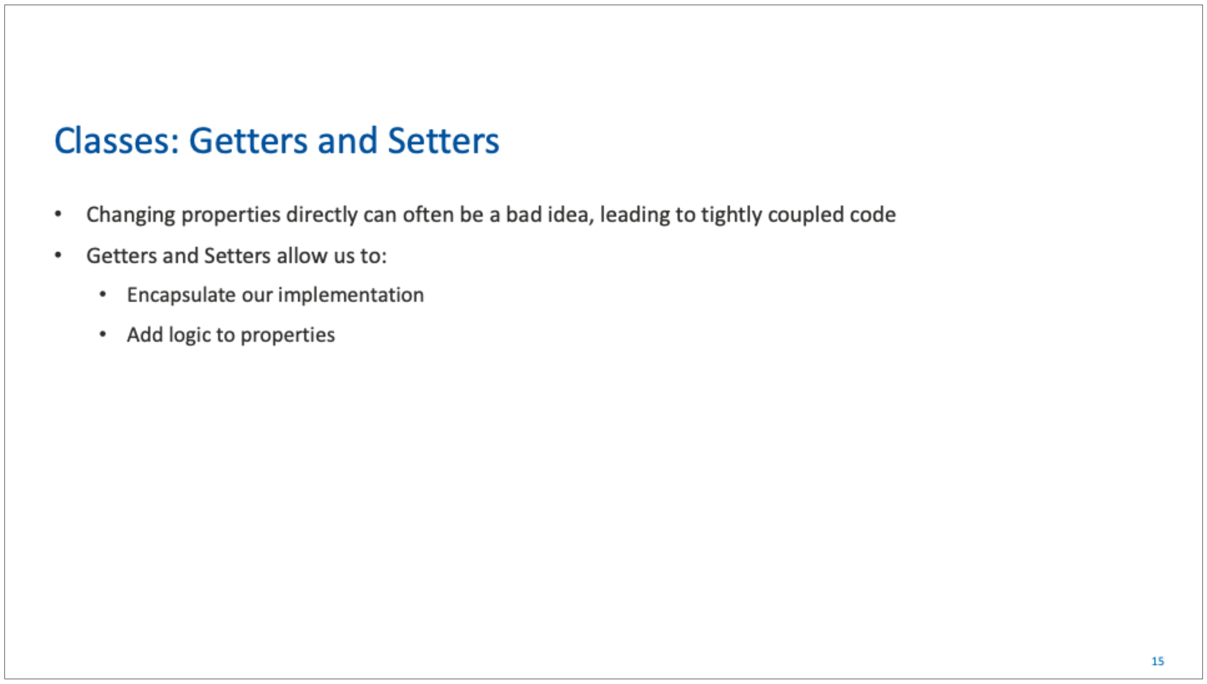
## CLASSES: PARAMETER PROPERTIES

```
class Vehicle {
  protected speed: number = 0;
  constructor (
    readonly wheels: number,
    readonly power: number
  ) {}
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

class Car extends Vehicle {
  readonly gps: Boolean = true;
  constructor (wheels, power) {
    super(wheels, power);
  }
}

let myCar = new Car(4, 20);
console.log(myCar.wheels); //4
```

Parameter properties stop us repeating ourselves quite so much by creating and initialising a property in one place.

By using a modifier in the parameter we create a property.

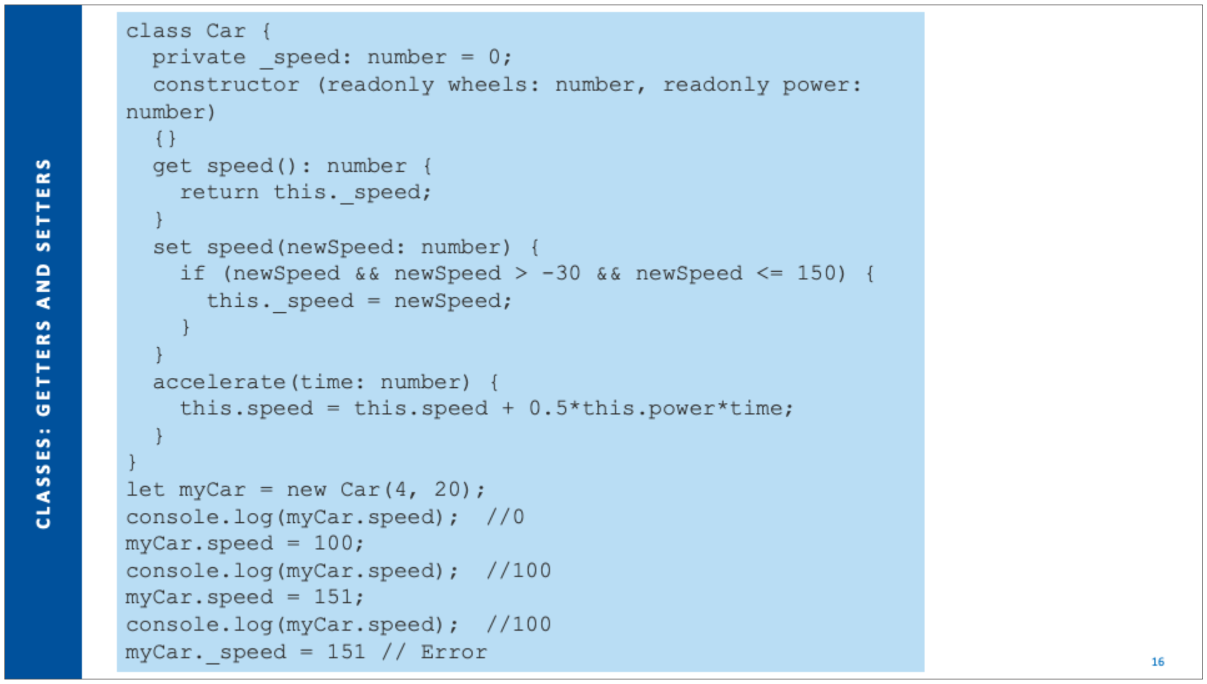
14



## Classes: Getters and Setters

- Changing properties directly can often be a bad idea, leading to tightly coupled code
- Getters and Setters allow us to:
  - Encapsulate our implementation
  - Add logic to properties

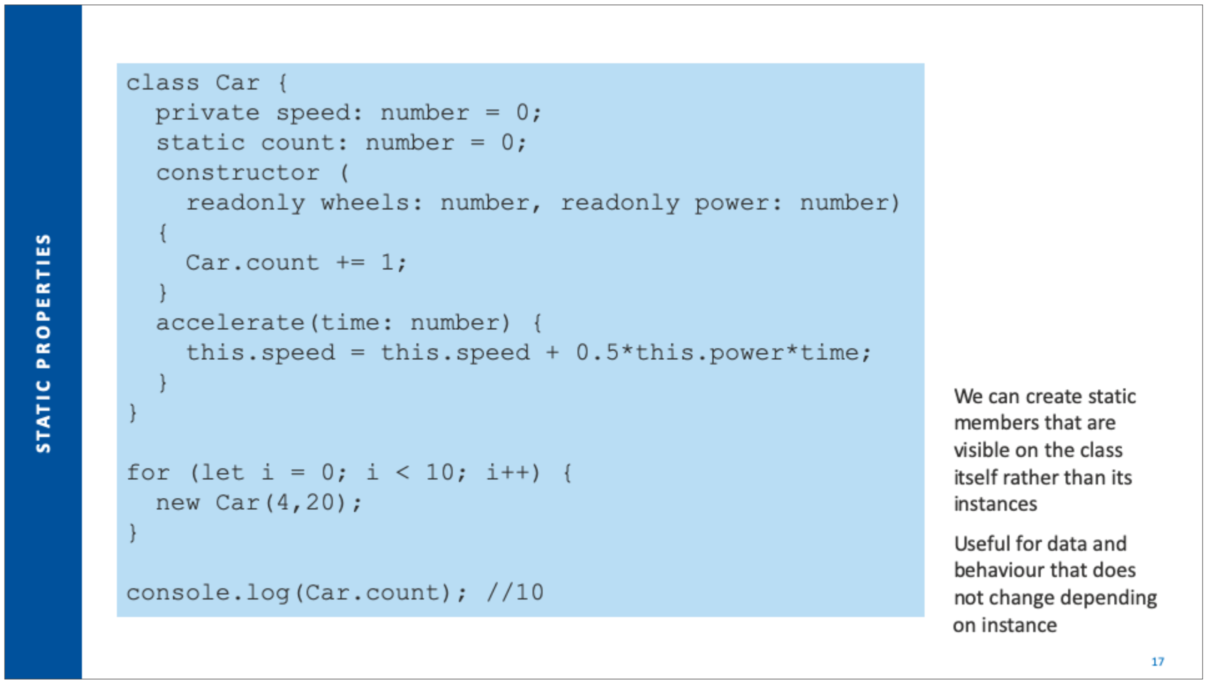
15



## CLASSES: GETTERS AND SETTERS

```
class Car {
  private _speed: number = 0;
  constructor (readonly wheels: number, readonly power:
number)
  {}
  get speed(): number {
    return this._speed;
  }
  set speed(newSpeed: number) {
    if (newSpeed && newSpeed > -30 && newSpeed <= 150) {
      this._speed = newSpeed;
    }
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}
let myCar = new Car(4, 20);
console.log(myCar.speed);  //0
myCar.speed = 100;
console.log(myCar.speed);  //100
myCar.speed = 151;
console.log(myCar.speed);  //100
myCar._speed = 151 // Error
```

16



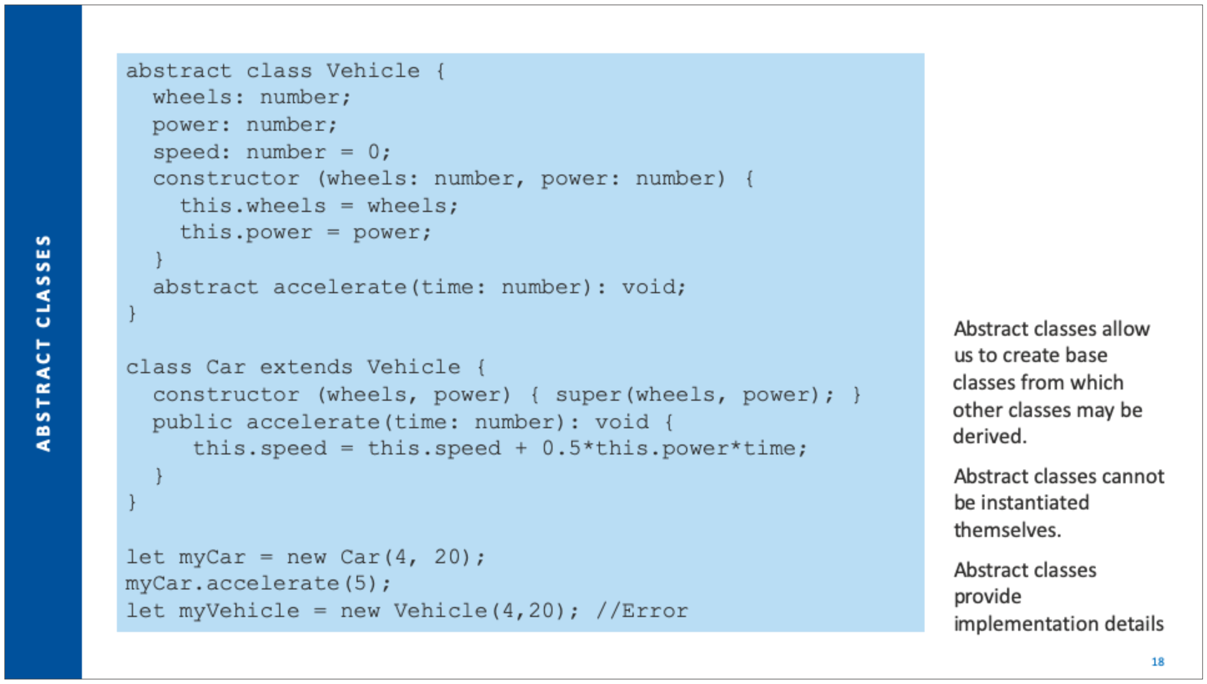
## STATIC PROPERTIES

```
class Car {
  private speed: number = 0;
  static count: number = 0;
  constructor (
    readonly wheels: number, readonly power: number)
  {
    Car.count += 1;
  }
  accelerate(time: number) {
    this.speed = this.speed + 0.5*this.power*time;
  }
}

for (let i = 0; i < 10; i++) {
  new Car(4,20);
}

console.log(Car.count); //10
```

We can create static members that are visible on the class itself rather than its instances

Useful for data and behaviour that does not change depending on instance

17



## ABSTRACT CLASSES

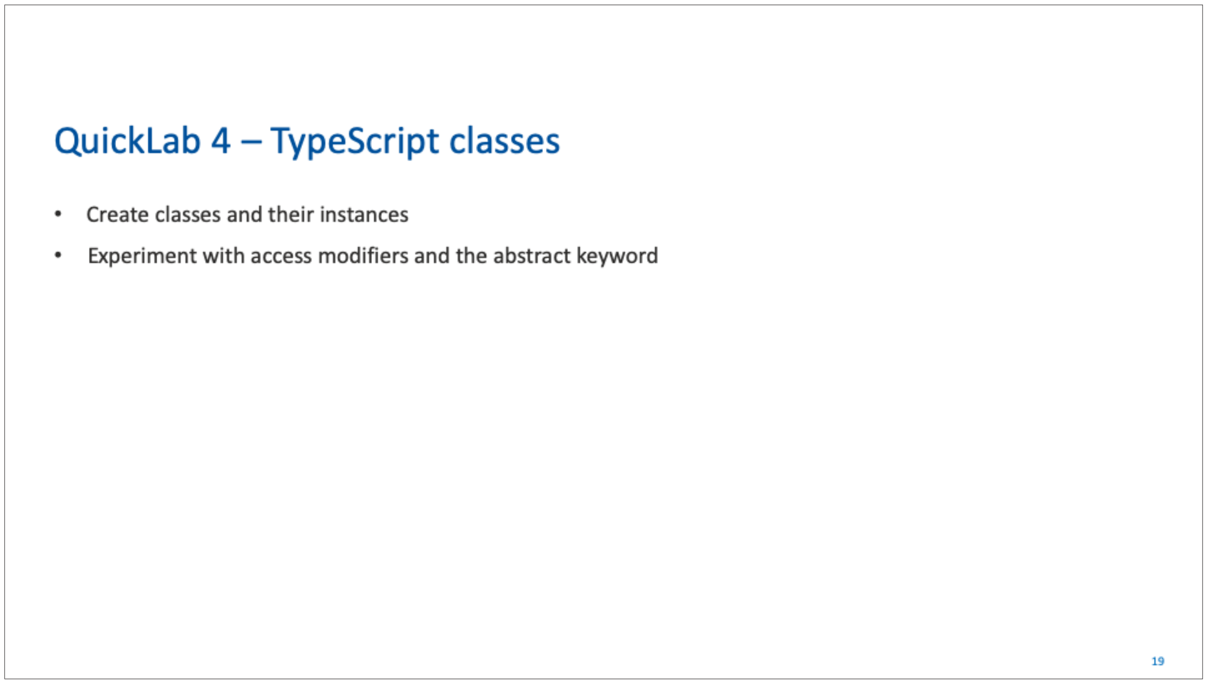
```
abstract class Vehicle {
  wheels: number;
  power: number;
  speed: number = 0;
  constructor (wheels: number, power: number) {
    this.wheels = wheels;
    this.power = power;
  }
  abstract accelerate(time: number): void;
}

class Car extends Vehicle {
  constructor (wheels, power) { super(wheels, power); }
  public accelerate(time: number): void {
      this.speed = this.speed + 0.5*this.power*time;
  }
}

let myCar = new Car(4, 20);
myCar.accelerate(5);
let myVehicle = new Vehicle(4,20); //Error
```

Abstract classes allow us to create base classes from which other classes may be derived.

Abstract classes cannot be instantiated themselves.

Abstract classes provide implementation details

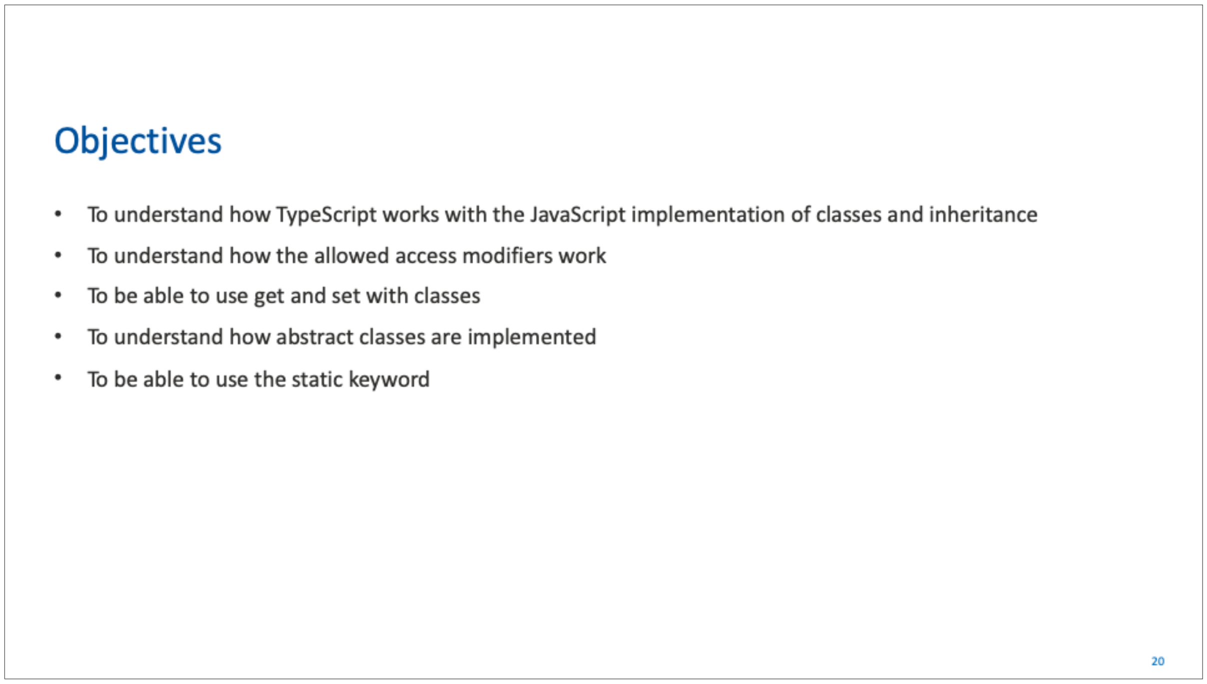
18



## QuickLab 4 – TypeScript classes

- Create classes and their instances
- Experiment with access modifiers and the abstract keyword

19



## Objectives

- To understand how TypeScript works with the JavaScript implementation of classes and inheritance
- To understand how the allowed access modifiers work
- To be able to use get and set with classes
- To understand how abstract classes are implemented
- To be able to use the static keyword

20