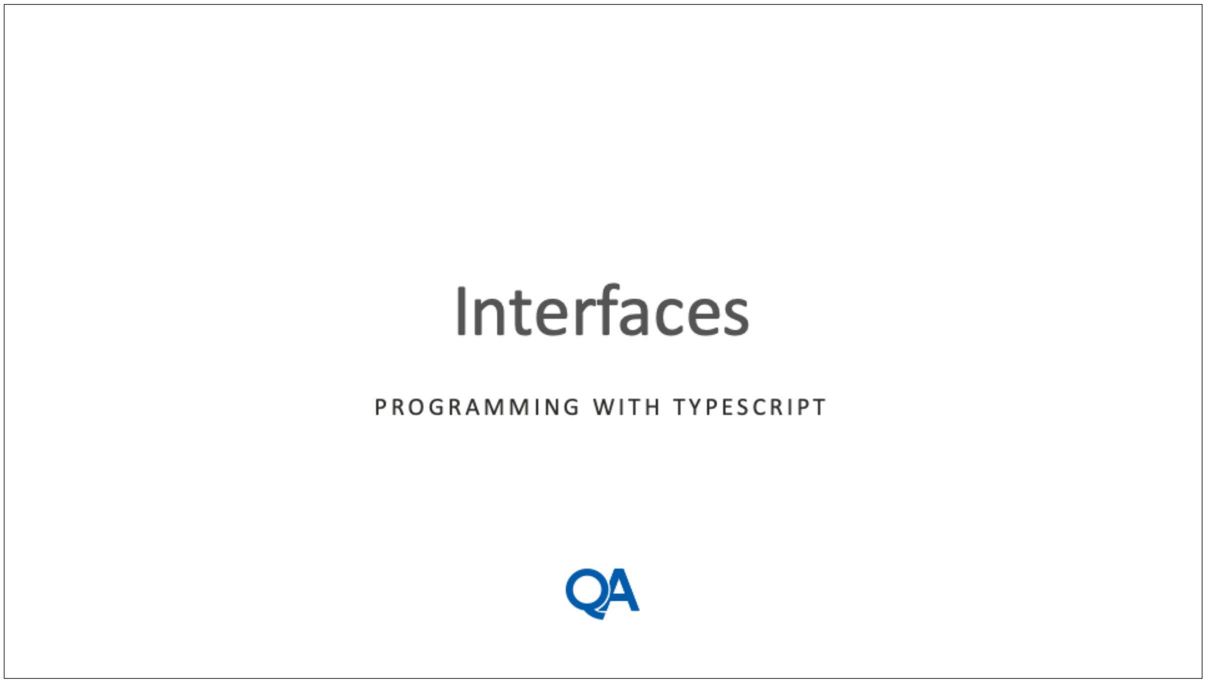
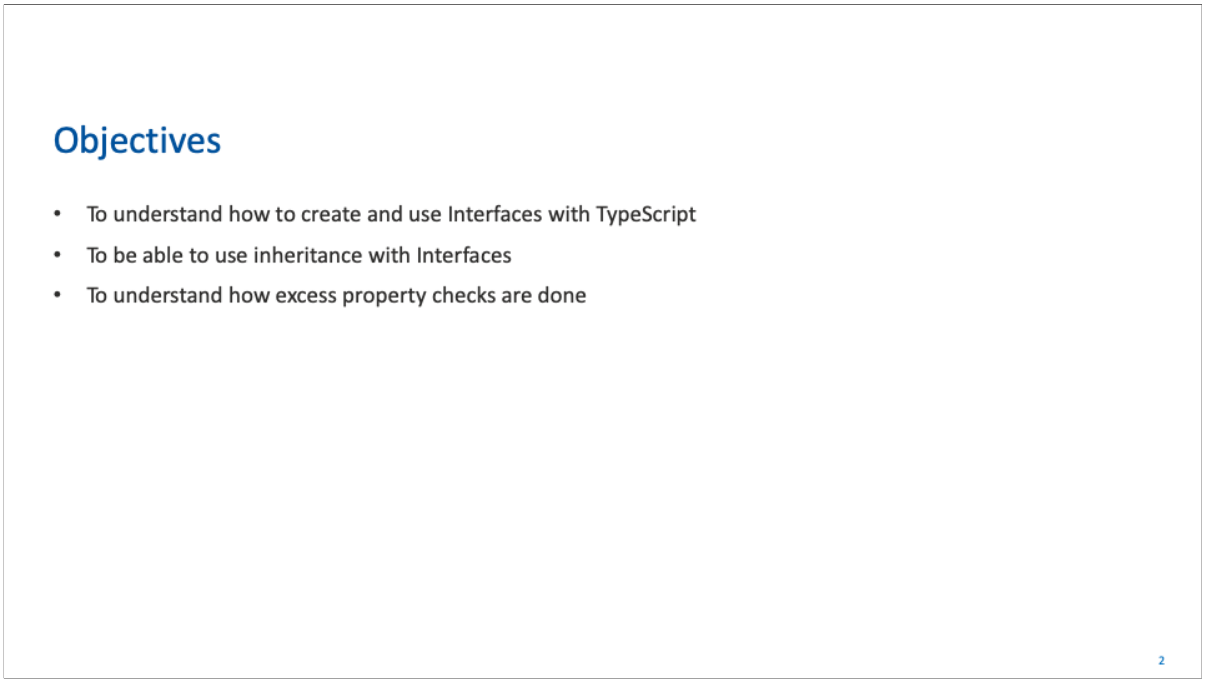


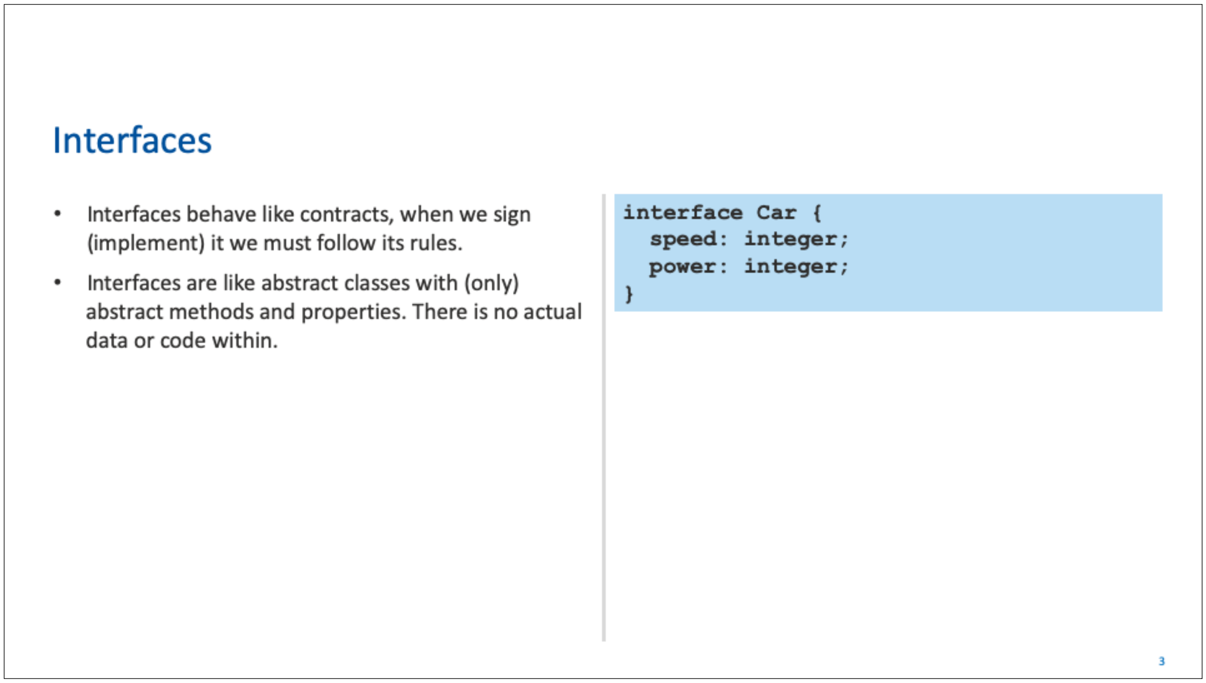
Interfaces

PROGRAMMING WITH TYPESCRIPT



Objectives

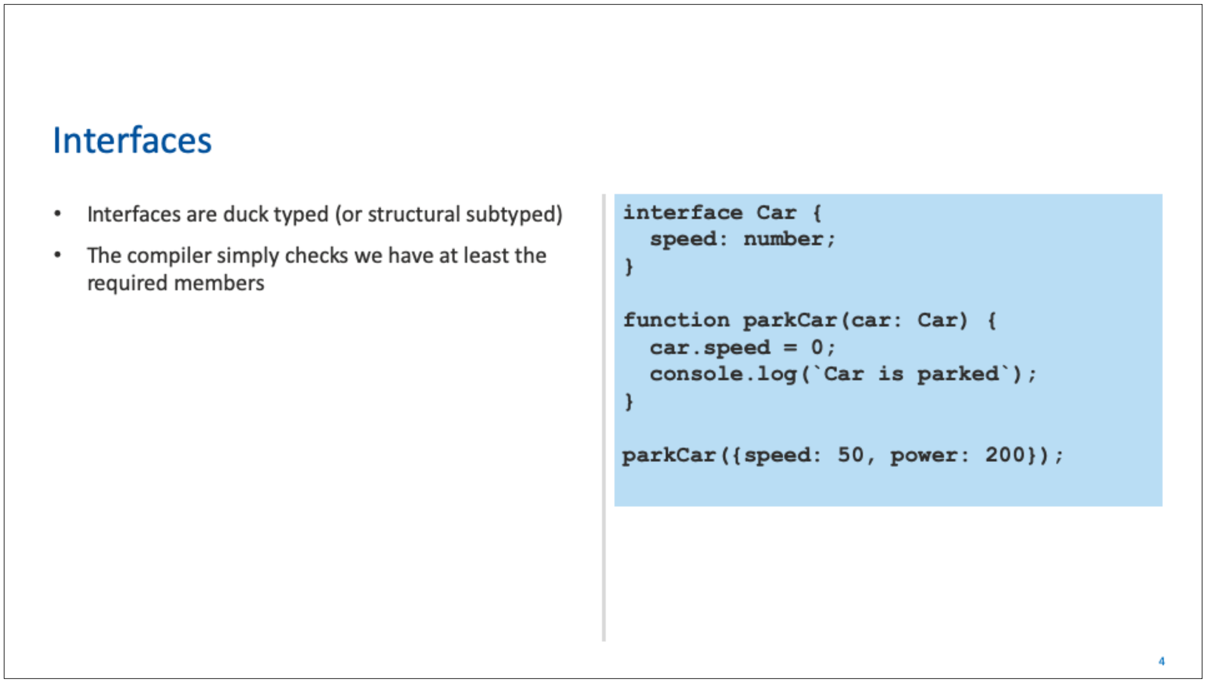
- To understand how to create and use Interfaces with TypeScript
- To be able to use inheritance with Interfaces
- To understand how excess property checks are done



Interfaces

- Interfaces behave like contracts, when we sign (implement) it we must follow its rules.
- Interfaces are like abstract classes with (only) abstract methods and properties. There is no actual data or code within.

```
interface Car {
  speed: integer;
  power: integer;
}
```

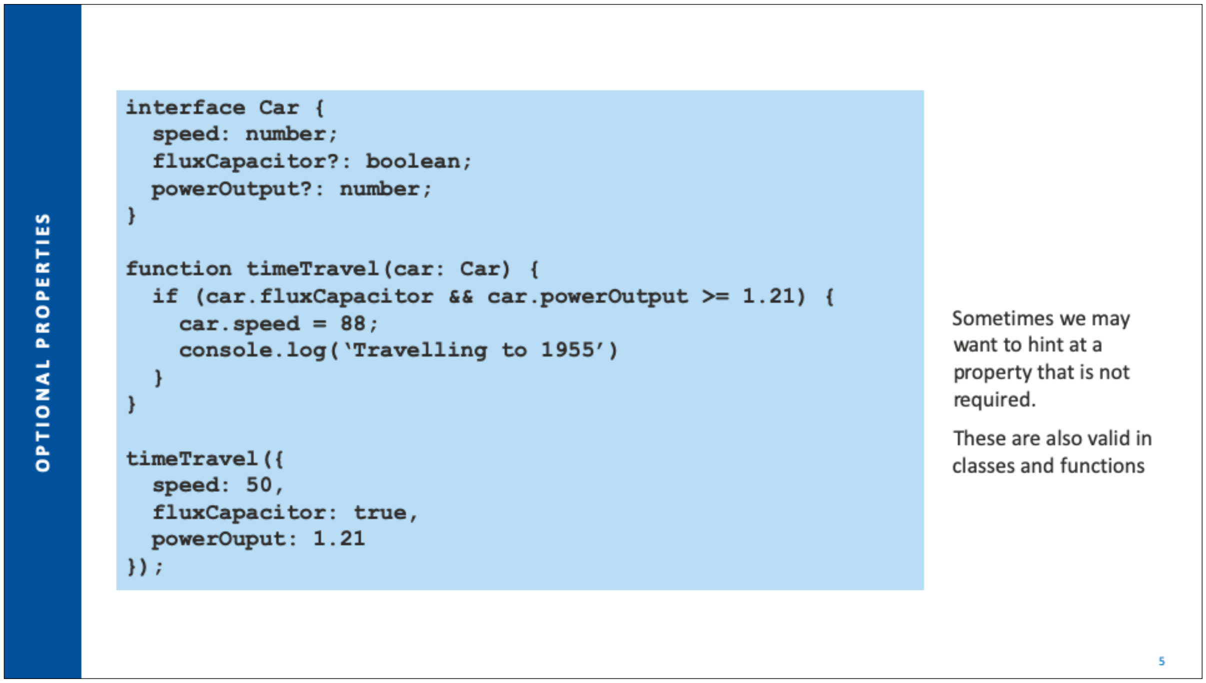


Interfaces

- Interfaces are duck typed (or structural subtyped)
- The compiler simply checks we have at least the required members

```
interface Car {
  speed: number;
}

function parkCar(car: Car) {
  car.speed = 0;
  console.log(`Car is parked`);
}

parkCar({speed: 50, power: 200});
```

4

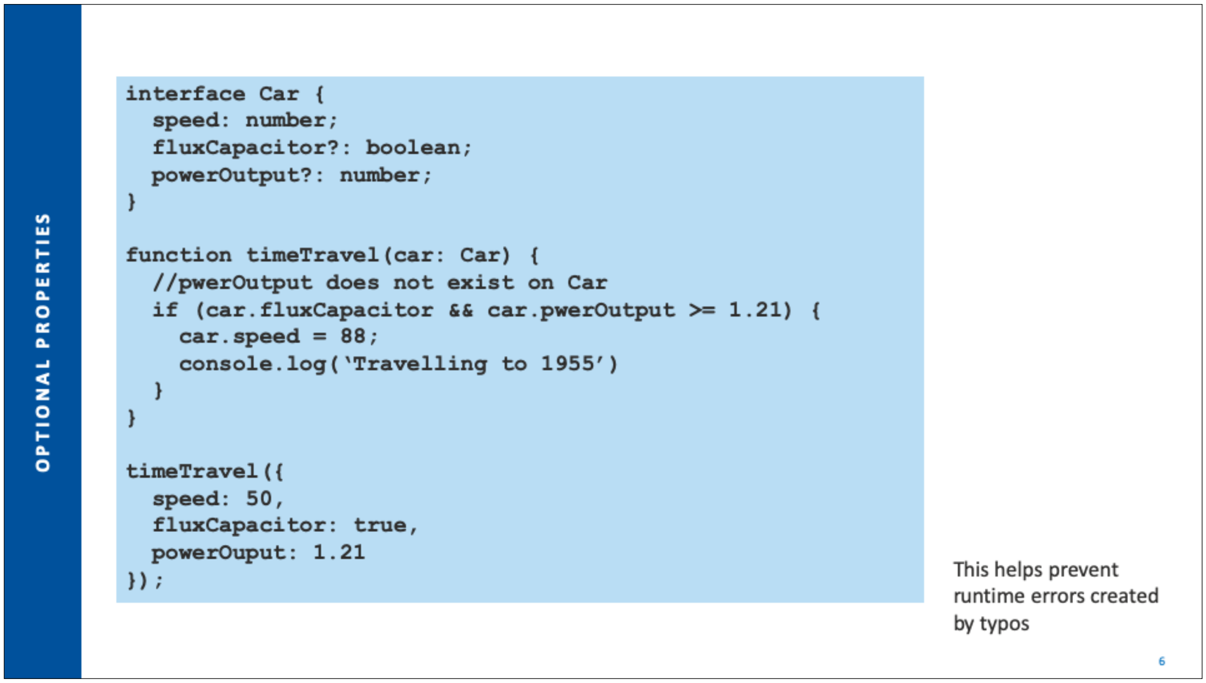


OPTIONAL PROPERTIES

```
interface Car {
  speed: number;
  fluxCapacitor?: boolean;
  powerOutput?: number;
}

function timeTravel(car: Car) {
  if (car.fluxCapacitor && car.powerOutput >= 1.21) {
    car.speed = 88;
    console.log(‘Travelling to 1955’)
  }
}

timeTravel({
  speed: 50,
  fluxCapacitor: true,
  powerOuput: 1.21
});
```

Sometimes we may want to hint at a property that is not required.

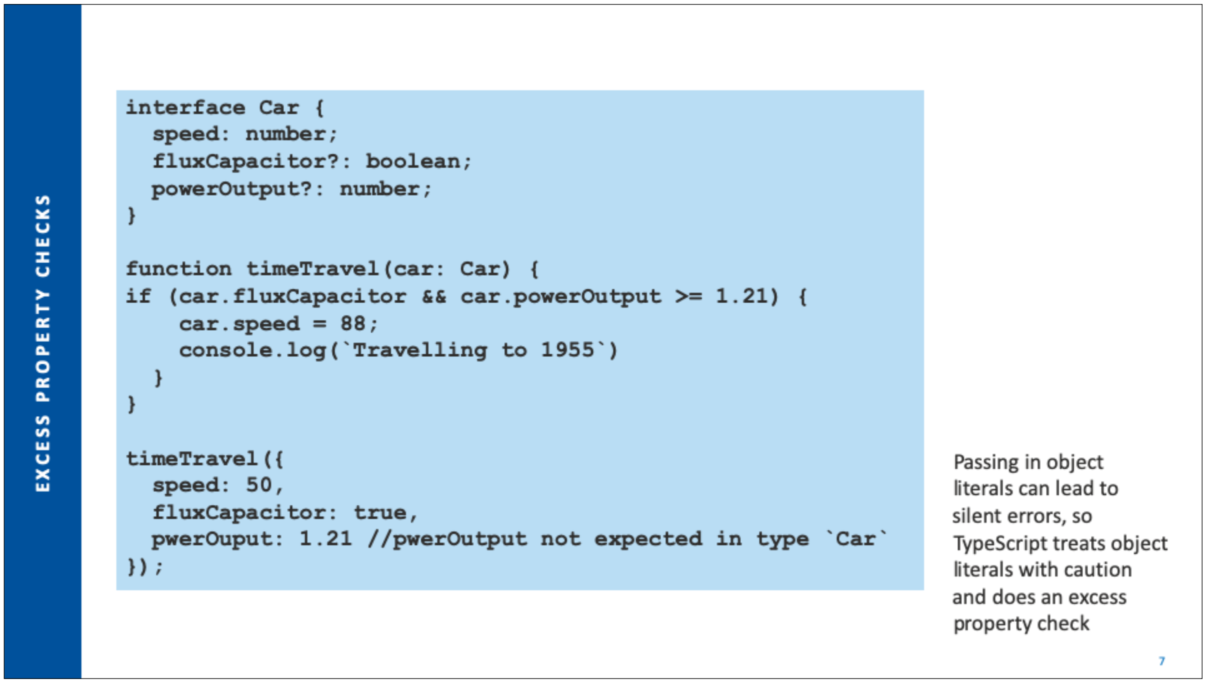
These are also valid in classes and functions



OPTIONAL PROPERTIES

```
interface Car {
  speed: number;
  fluxCapacitor?: boolean;
  powerOutput?: number;
}

function timeTravel(car: Car) {
  //pwerOutput does not exist on Car
  if (car.fluxCapacitor && car.pwerOutput >= 1.21) {
    car.speed = 88;
    console.log(‘Travelling to 1955’)
  }
}

timeTravel({
  speed: 50,
  fluxCapacitor: true,
  powerOuput: 1.21
});
```

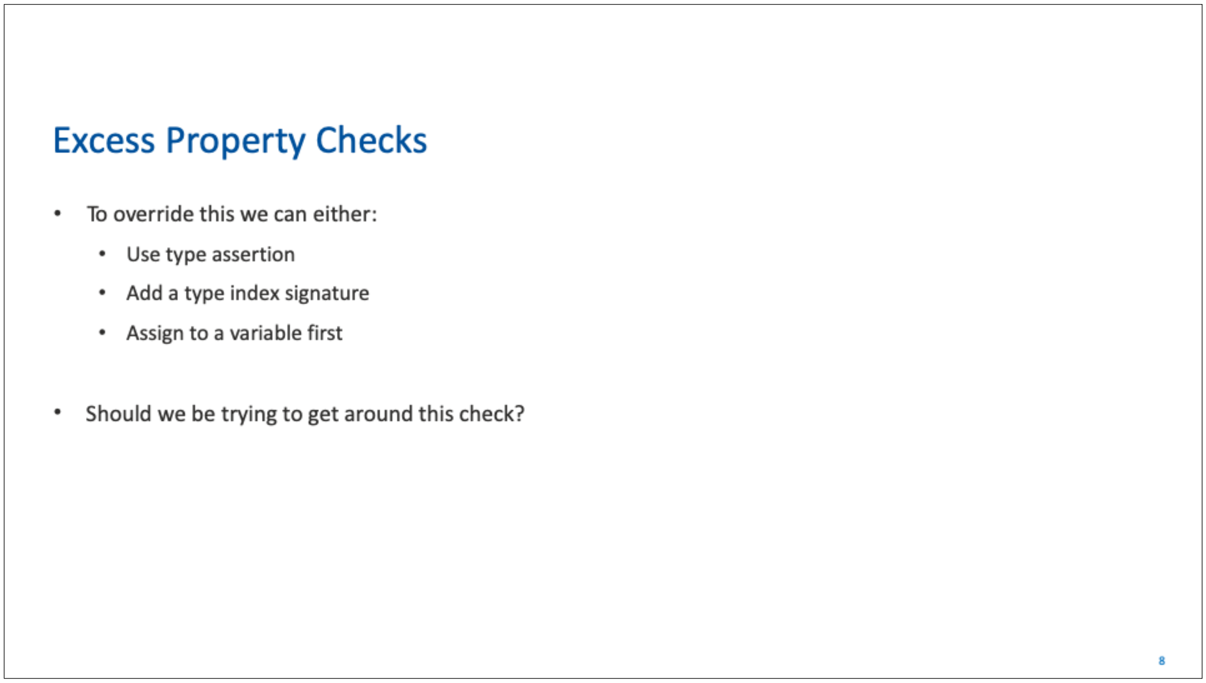
This helps prevent runtime errors created by typos



EXCESS PROPERTY CHECKS

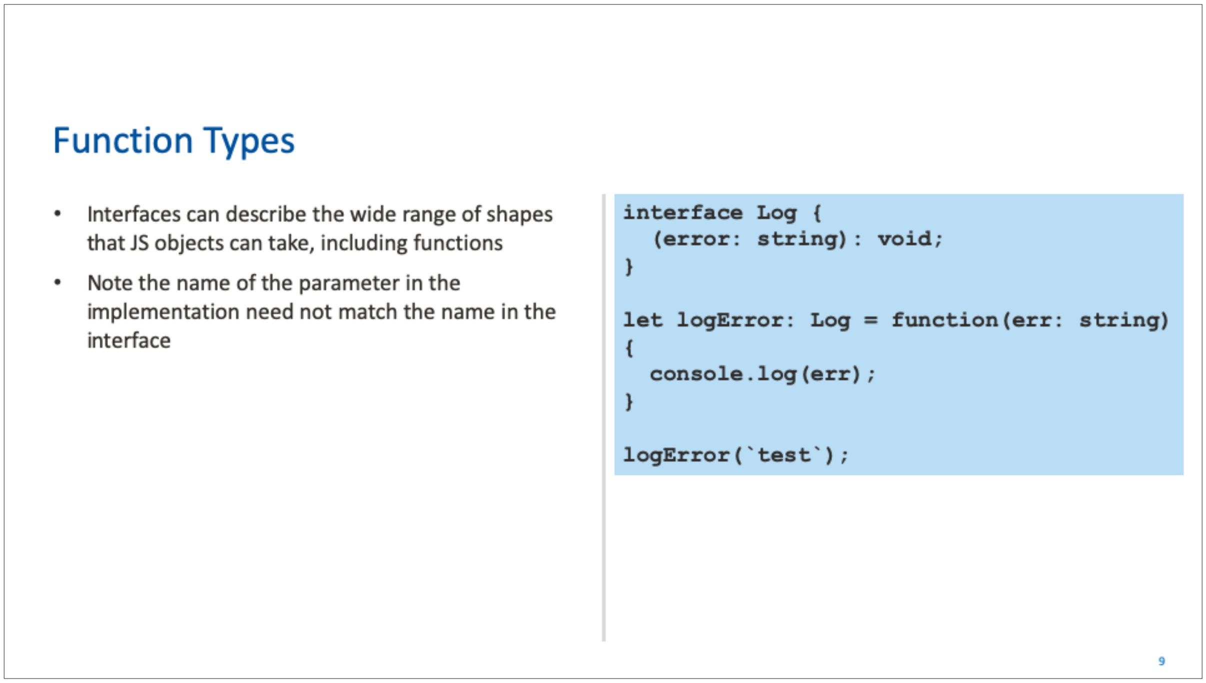
```
interface Car {
  speed: number;
  fluxCapacitor?: boolean;
  powerOutput?: number;
}

function timeTravel(car: Car) {
if (car.fluxCapacitor && car.powerOutput >= 1.21) {
    car.speed = 88;
    console.log(`Travelling to 1955`)
  }
}

timeTravel({
  speed: 50,
  fluxCapacitor: true,
  pwerOuput: 1.21 //pwerOutput not expected in type `Car`
});
```

Passing in object literals can lead to silent errors, so TypeScript treats object literals with caution and does an excess property check



Excess Property Checks

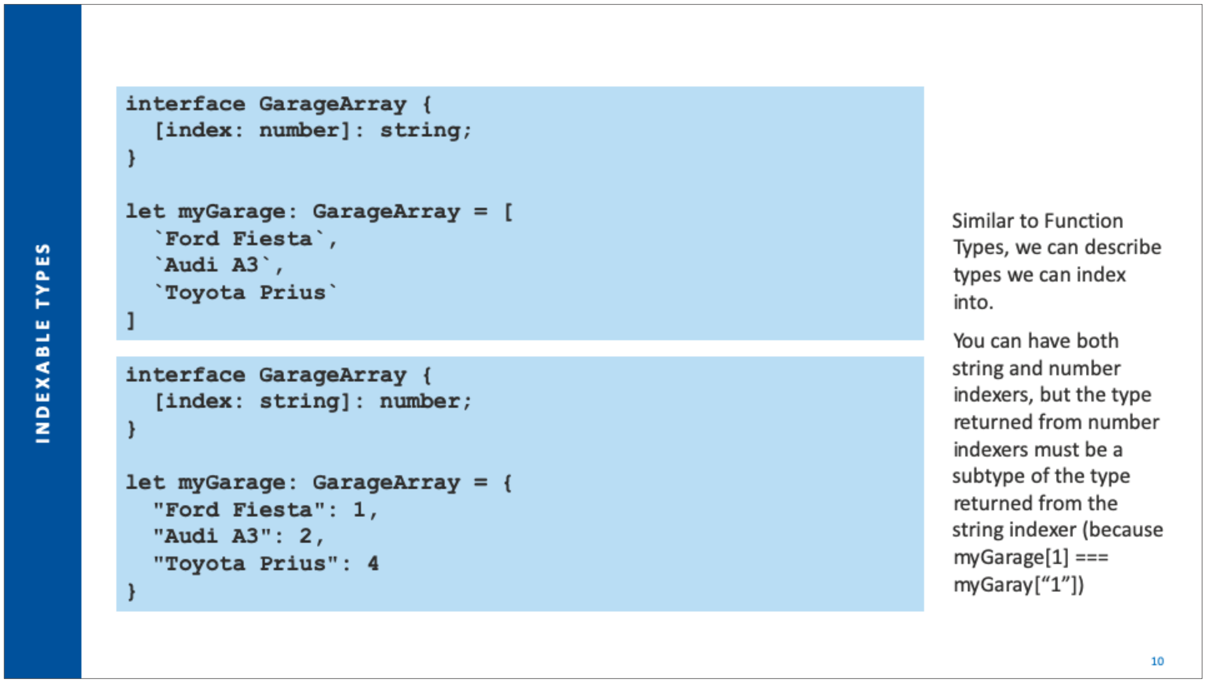
- To override this we can either:
  - Use type assertion
  - Add a type index signature
  - Assign to a variable first
- Should we be trying to get around this check?



Function Types

- Interfaces can describe the wide range of shapes that JS objects can take, including functions
- Note the name of the parameter in the implementation need not match the name in the interface

```
interface Log {
  (error: string): void;
}

let logError: Log = function(err: string)
{
  console.log(err);
}

logError(`test`);
```



INDEXABLE TYPES

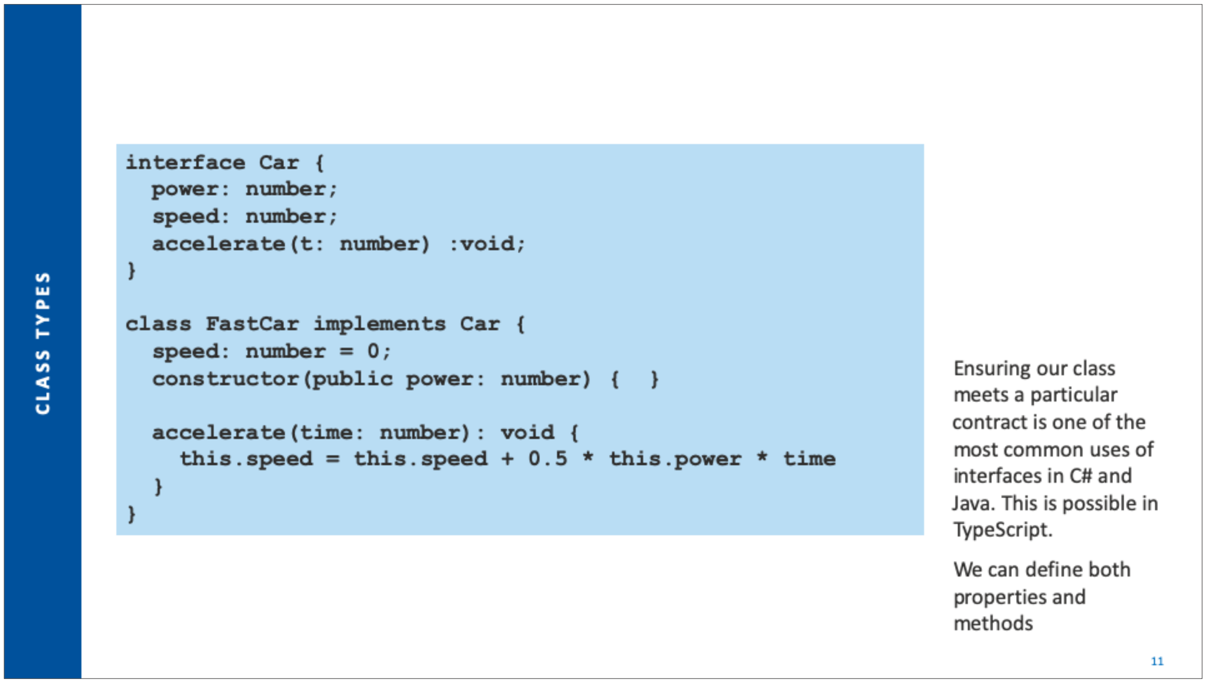
```
interface GarageArray {
  [index: number]: string;
}

let myGarage: GarageArray = [
  `Ford Fiesta`,
  `Audi A3`,
  `Toyota Prius`
]
```

```
interface GarageArray {
  [index: string]: number;
}

let myGarage: GarageArray = {
  "Ford Fiesta": 1,
  "Audi A3": 2,
  "Toyota Prius": 4
}
```

Similar to Function Types, we can describe types we can index into.

You can have both string and number indexers, but the type returned from number indexers must be a subtype of the type returned from the string indexer (because myGarage[1] === myGaray["1"])

10



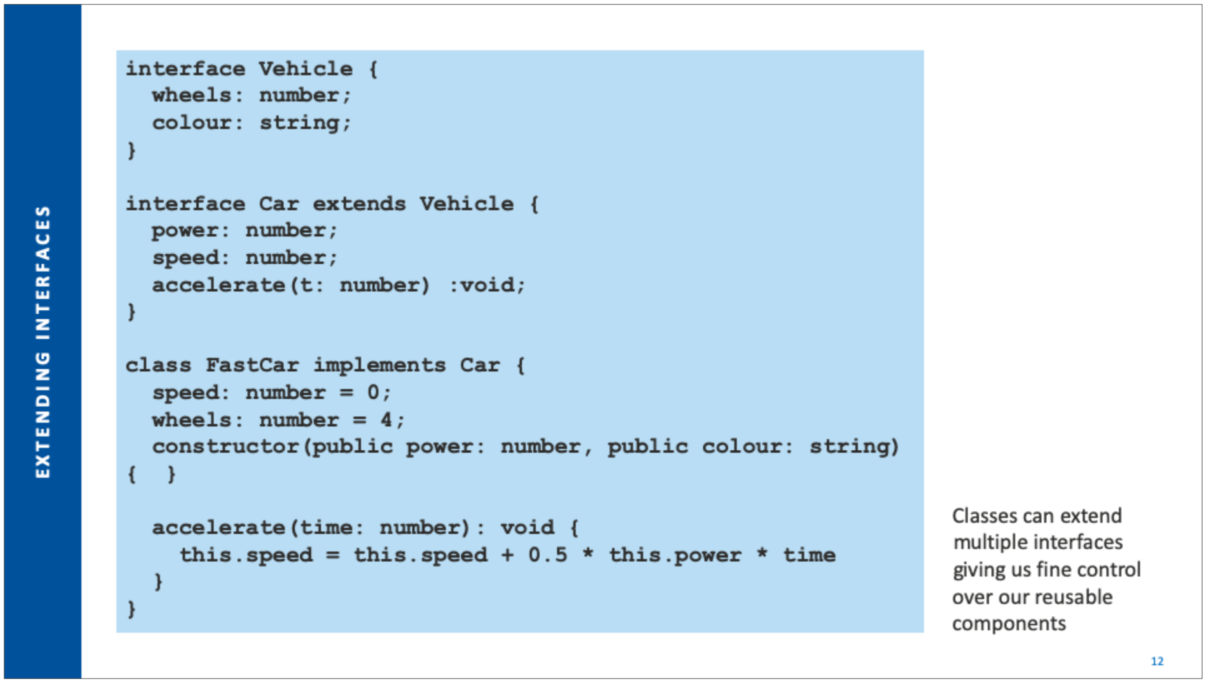
CLASS TYPES

```
interface Car {
  power: number;
  speed: number;
  accelerate(t: number) :void;
}

class FastCar implements Car {
  speed: number = 0;
  constructor(public power: number) {   }

  accelerate(time: number): void {
    this.speed = this.speed + 0.5 * this.power * time
  }
}
```

Ensuring our class meets a particular contract is one of the most common uses of interfaces in C# and Java. This is possible in TypeScript.

We can define both properties and methods

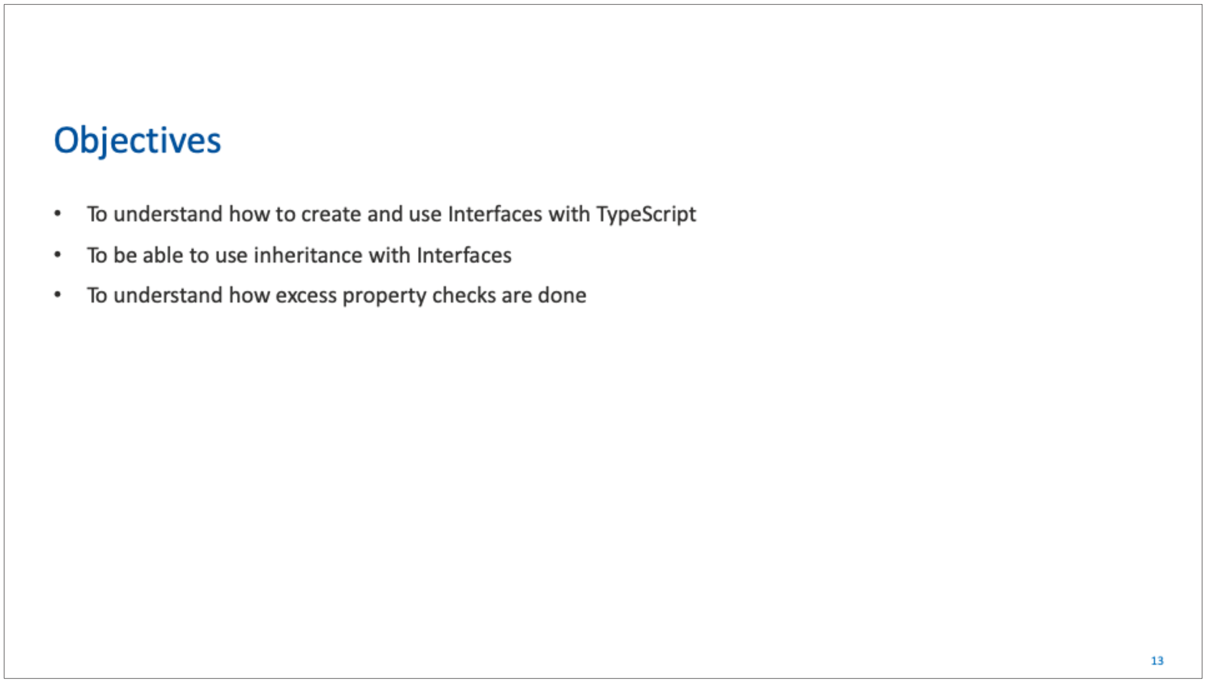
11



EXTENDING INTERFACES

```
interface Vehicle {
  wheels: number;
  colour: string;
}

interface Car extends Vehicle {
  power: number;
  speed: number;
  accelerate(t: number) :void;
}

class FastCar implements Car {
  speed: number = 0;
  wheels: number = 4;
  constructor(public power: number, public colour: string)
{  }

  accelerate(time: number): void {
    this.speed = this.speed + 0.5 * this.power * time
  }
}
```

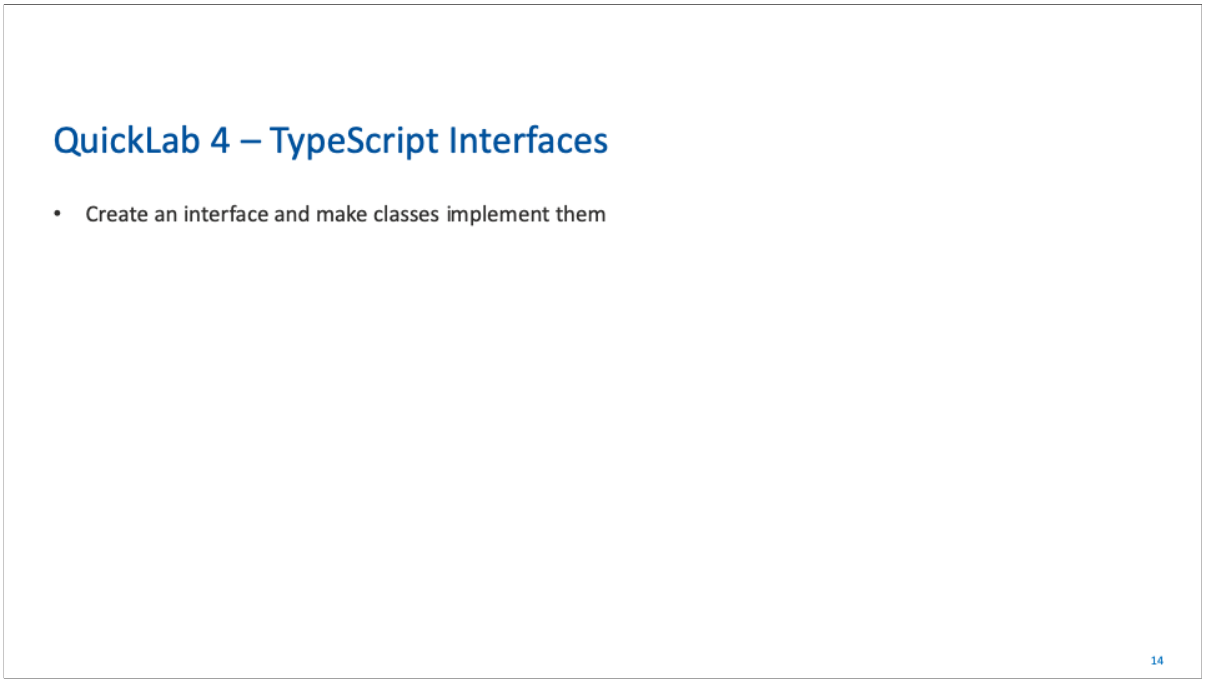
Classes can extend multiple interfaces giving us fine control over our reusable components



Objectives

- To understand how to create and use Interfaces with TypeScript
- To be able to use inheritance with Interfaces
- To understand how excess property checks are done

13



QuickLab 4 – TypeScript Interfaces

- Create an interface and make classes implement them