Typescript TS

Inheritance, Getter & Setter

Inheritance in TypeScript

- **Inheritance** refers to the ability of an object to **reuse the properties and methods of other objects**.
- The main purpose of the inheritance is to **re-use code**.
- Inheritance in TypeScript enables you to create a **new class by using the existing class.**
- The new class will **reuse** the properties and methods of the existing class.
- It may also modify the behavior of the existing class by adding its **own properties and methods**.
- Inheritance can be **single-level** or **multi-level.**
- But a Class can extend only from a **single Class**.

Creating Inheritance in TypeScript

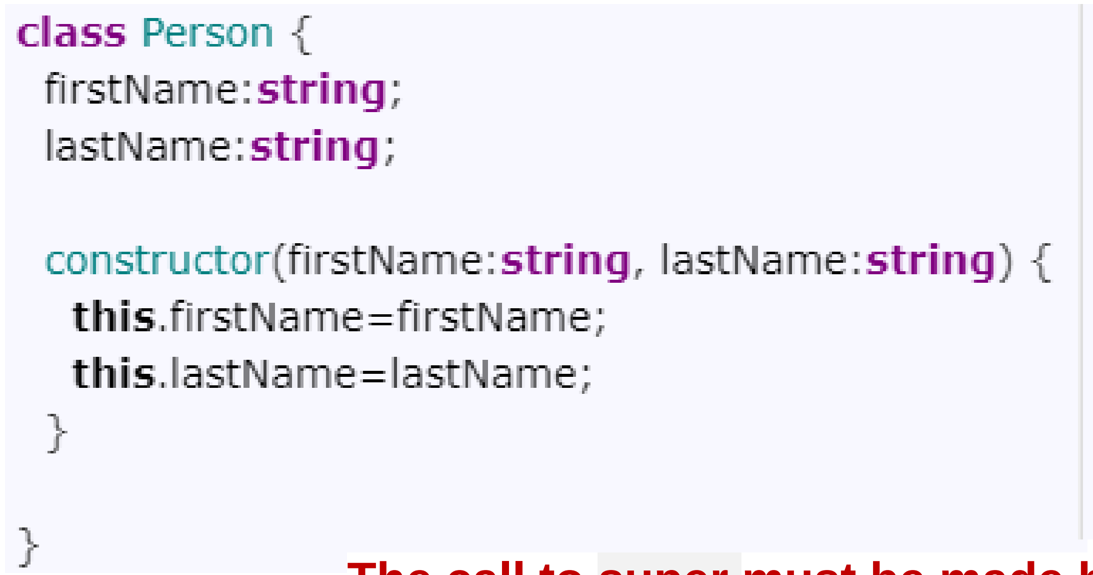
- To create inheritance, we use the **extends** keyword.

```
class <ClassName> extends <ParentClassName>
{
      // methods and fields
{
```

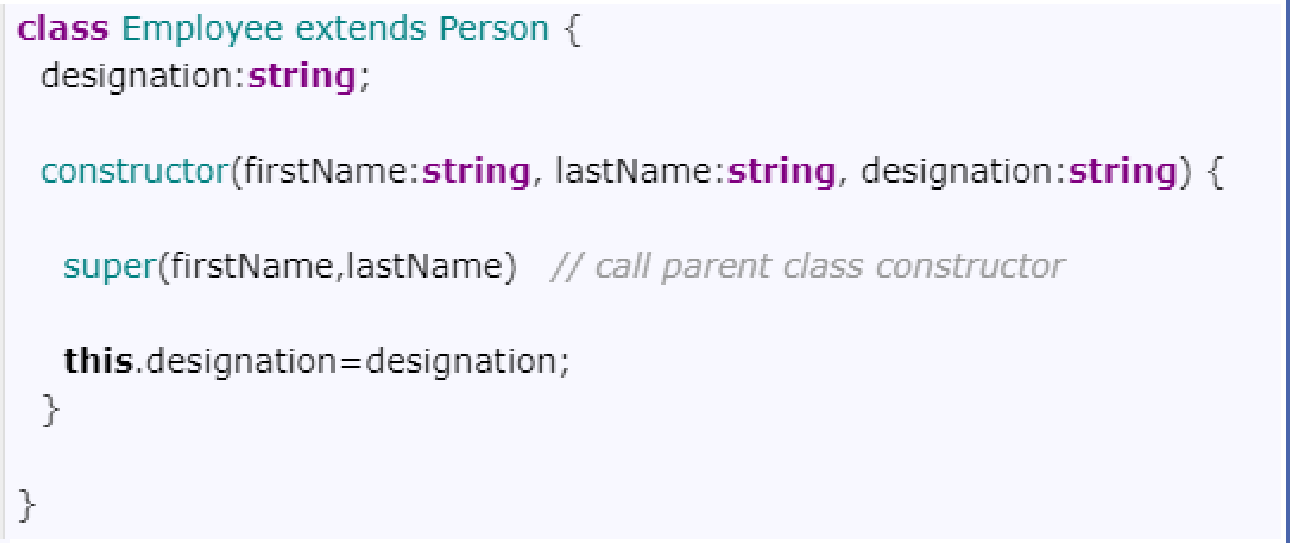
```
class Person {
  firstName:string="";
  lastName:string="";

  getName() :string {
    return this.firstName+ " "+this.lastName
  }
}


class Employee extends Person{
  designation:string="";
}

//extend class gets all the properties and methods of the Parent Class
let e= new Employee();
e.firstName="Jon"
e.lastName="Snow"
e.designation="Lead Cast"
console.log(e.getName())  //"Jon Snow"
console.log(e)  // Employee: {  "firstName": "Jon",  "lastName": "Snow",  "designation": "Lead Cast"}
```

Invoking Parent class constructor

- In the previous example, we used the simple class without a constructor.
- If the parent class has a constructor, then the child class must invoke the parent class constructor in its constructor. We do that by using a special method super.



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

  constructor(firstName:string, lastName:string) {
    this.firstName=firstName;
    this.lastName=lastName;
  }

}
```

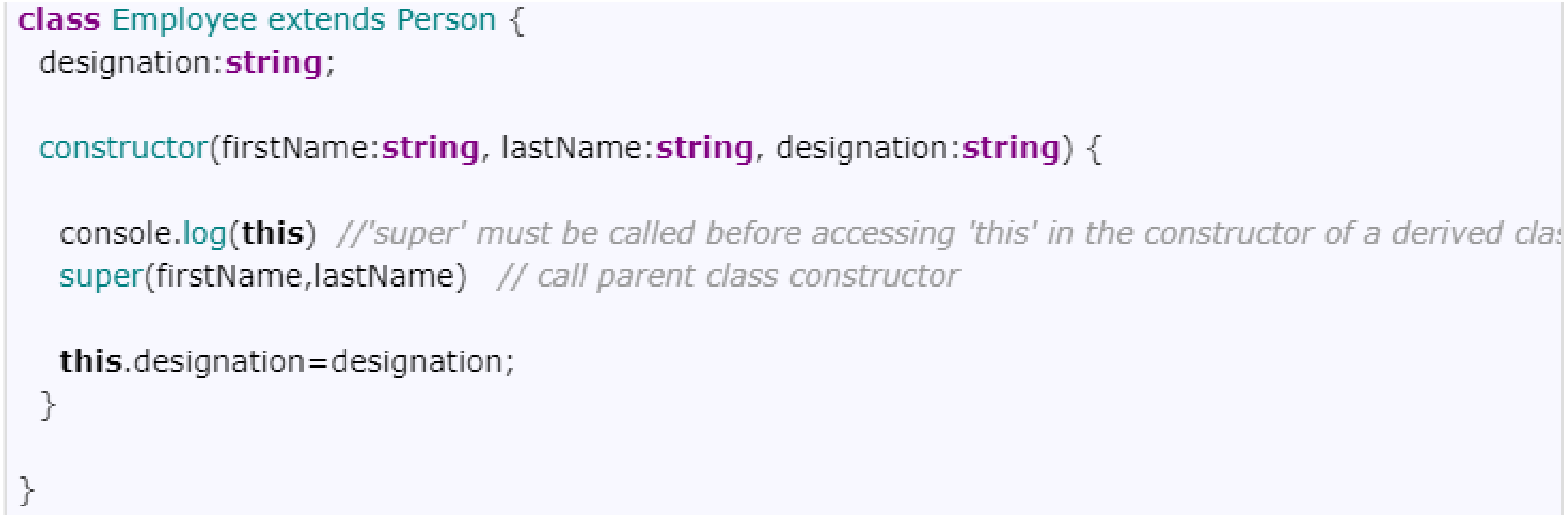


```
class Employee extends Person {
  designation:string;

  constructor(firstName:string, lastName:string, designation:string) {

    super(firstName,lastName)   // call parent class constructor

    this.designation=designation;
  }

}
```

The call to super must be made before the use of ‘this’ keyword.

Invoking Parent class constructor

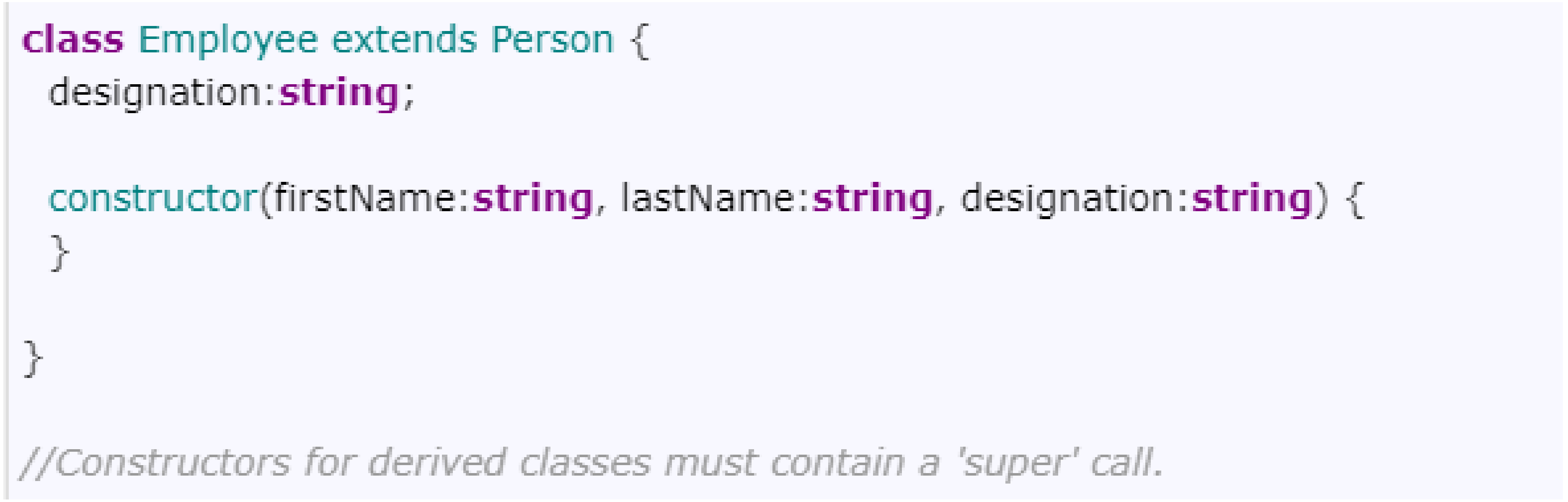
- In this example, we try to access the this **before we call the super** method. **The compiler flags this as an error (“*‘super’ must be called before accessing ‘this’ in the constructor of a derived class*“).**



```
class Employee extends Person {
  designation:string;

  constructor(firstName:string, lastName:string, designation:string) {

    console.log(this)  //'super' must be called before accessing 'this' in the constructor of a derived clas
    super(firstName,lastName)   // call parent class constructor

    this.designation=designation;
  }

}
```

Invoking Parent class constructor

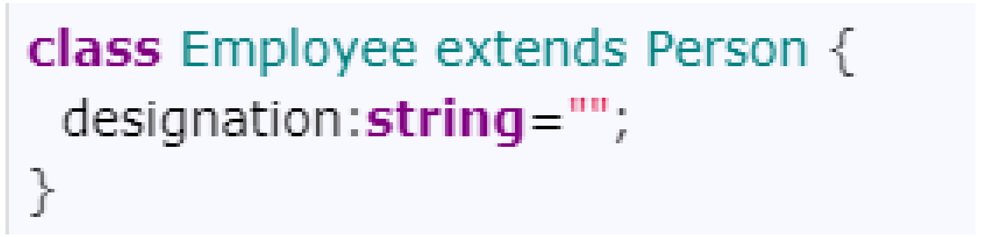
- If we do not call the super method, then the compiler flag this as an ("*Constructors for derived classes must contain a 'super' call*") error.



```
class Employee extends Person {
  designation:string;

  constructor(firstName:string, lastName:string, designation:string) {
  }

}

//Constructors for derived classes must contain a 'super' call.
```

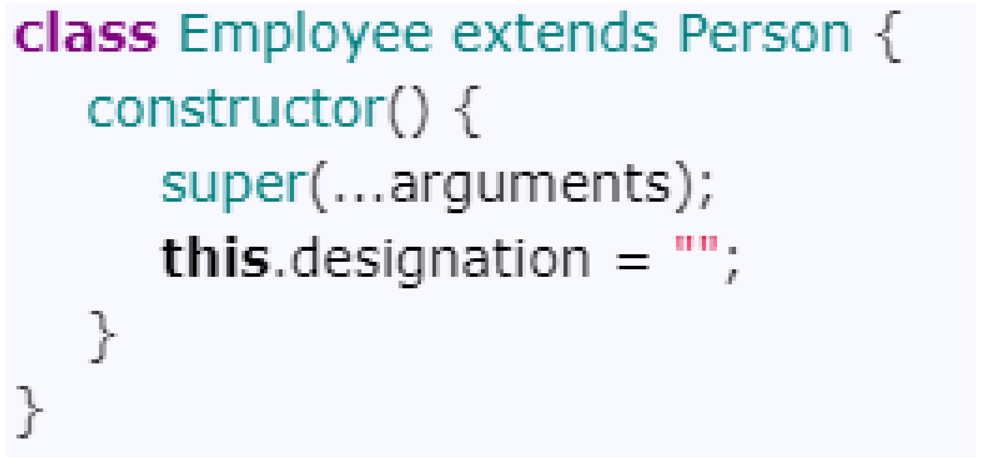
- If the child class does not need a constructor, then you can omit to call super method. The compiler will not flag this as an error.

Invoking Parent class constructor



```
class Employee extends Person {
 designation:string="";
}
```

- But the Typescript automatically creates a constructor method with a super call in the generated JavaScript. The above results in the following JavaScript code.



```
class Employee extends Person {
   constructor() {
      super(...arguments);
      this.designation = "";
   }
}
```

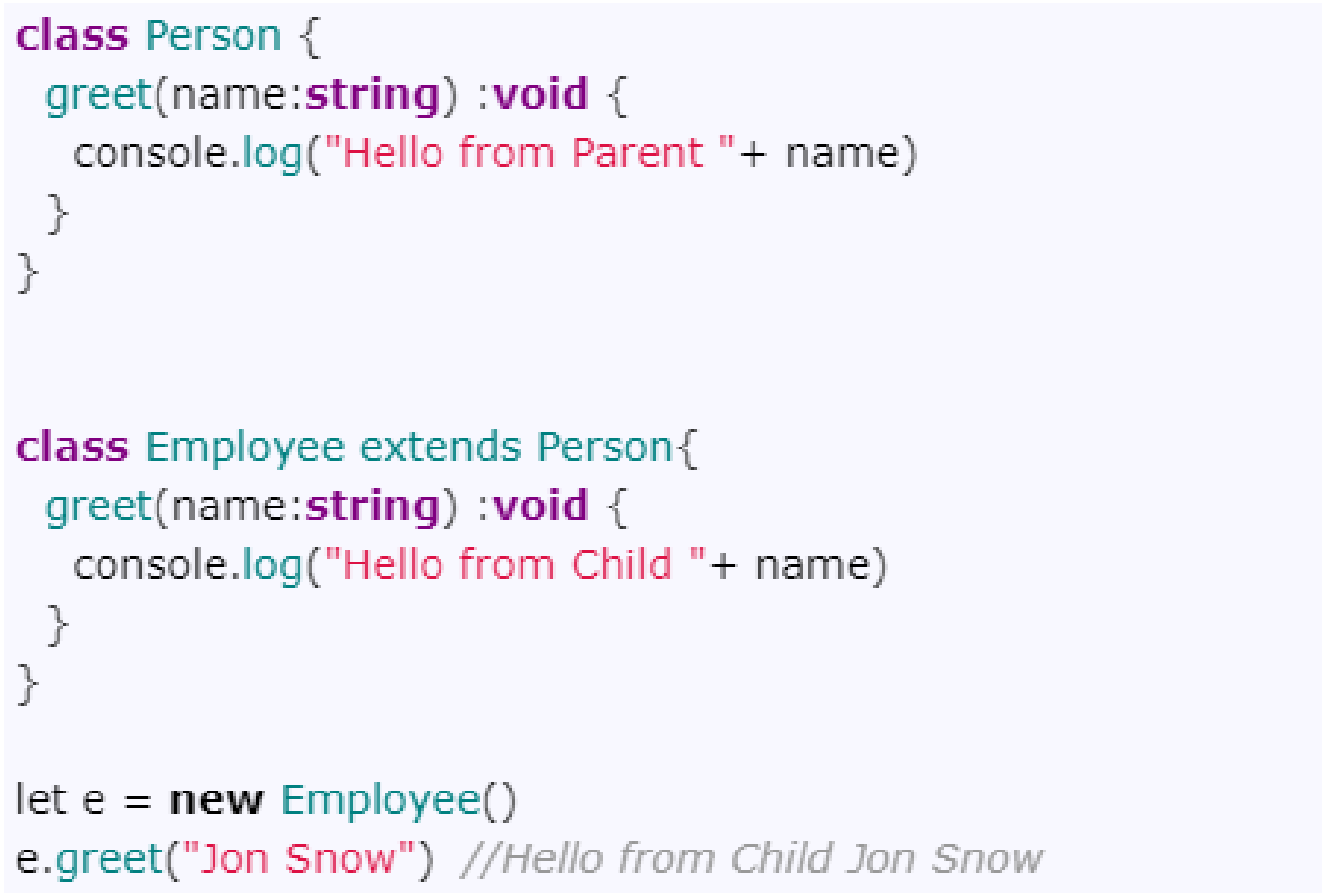
Summary

- The call to super must be made before the use of **'this'** keyword.
- If the derived class has a constructor method, then calling super is compulsory.
- In case the parent class does not have a constructor method, then invoke super without any parameter.
- If both parent and child do not need a constructor method, then you do not have to create one. JavaScript will automatically create an empty constructor for both.

Method Overriding

- You can override the Methods & Properties of the Parent class in the child class.
- In method overriding the types of the parameters and the return type of the method have to be **compatible** with that of the parent class.
- While in Property overriding the data type of the Property have to be **compatible** with that of the parent class.

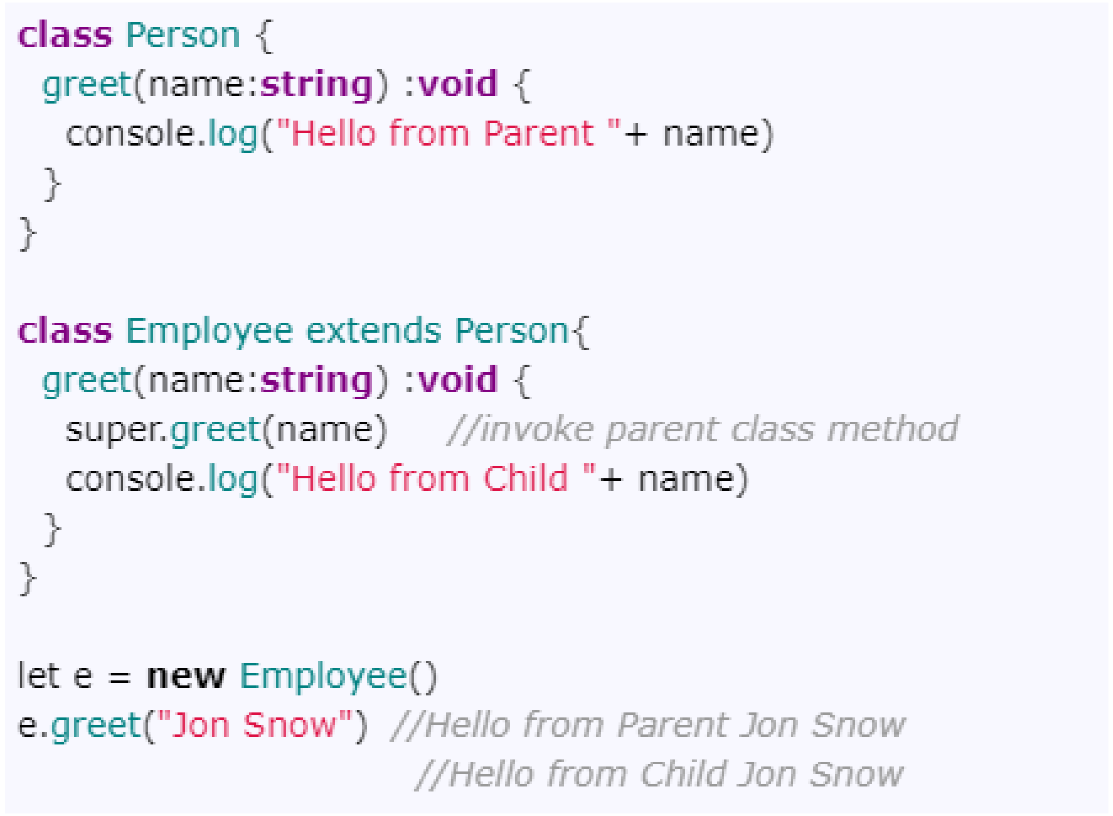
Method Overriding



```
class Person {
  greet(name:string) :void {
    console.log("Hello from Parent "+ name)
  }
}


class Employee extends Person{
  greet(name:string) :void {
    console.log("Hello from Child "+ name)
  }
}

let e = new Employee()
e.greet("Jon Snow")  //Hello from Child Jon Snow
```

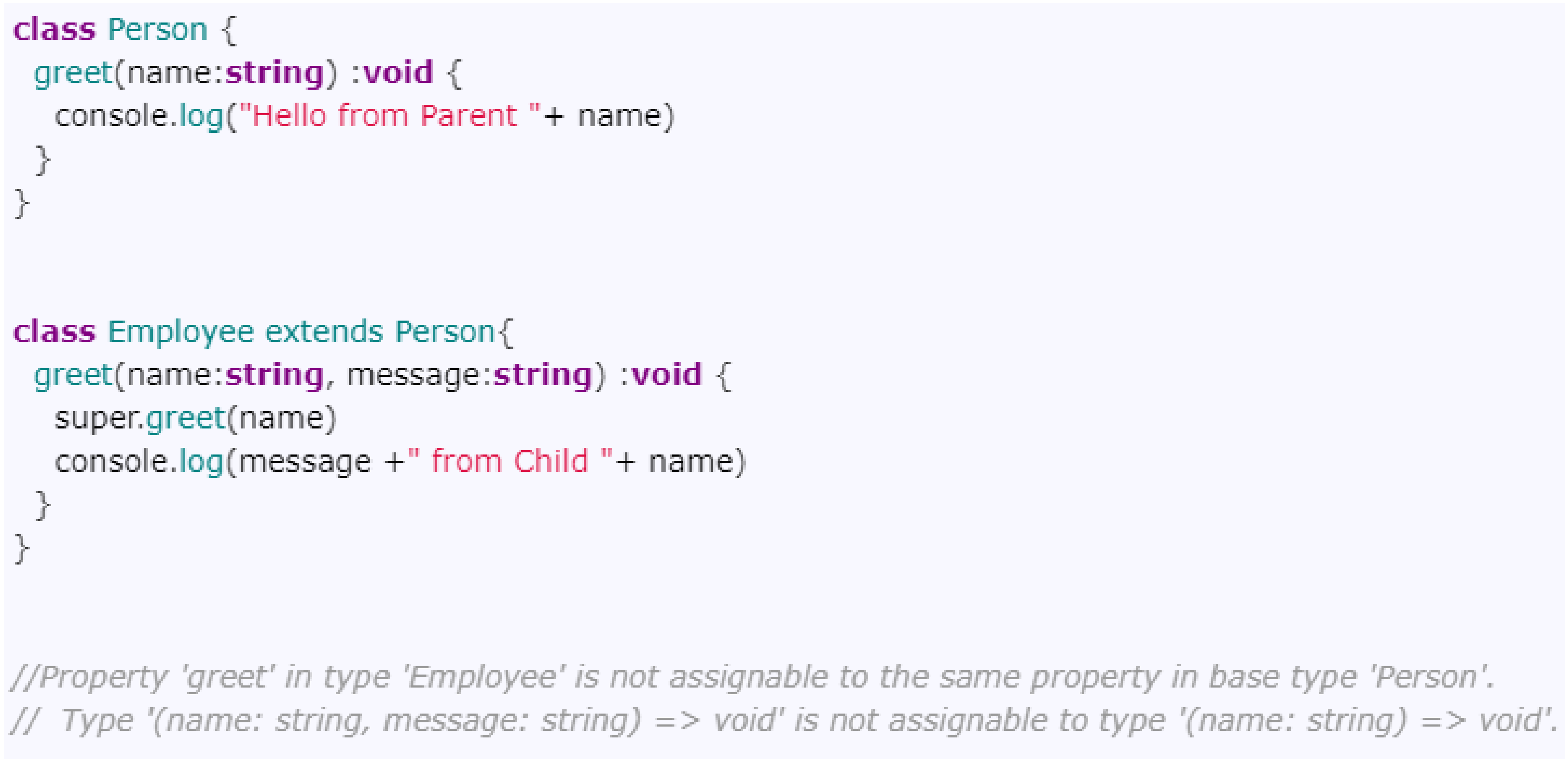
Method Overriding

- You can use the super to invoke the parent class method. In this example super.greet() method invokes the greet method from the parent class.



```
class Person {
  greet(name:string) :void {
    console.log("Hello from Parent "+ name)
  }
}

class Employee extends Person{
  greet(name:string) :void {
    super.greet(name)     //invoke parent class method
    console.log("Hello from Child "+ name)
  }
}

let e = new Employee()
e.greet("Jon Snow")  //Hello from Parent Jon Snow
                      //Hello from Child Jon Snow
```

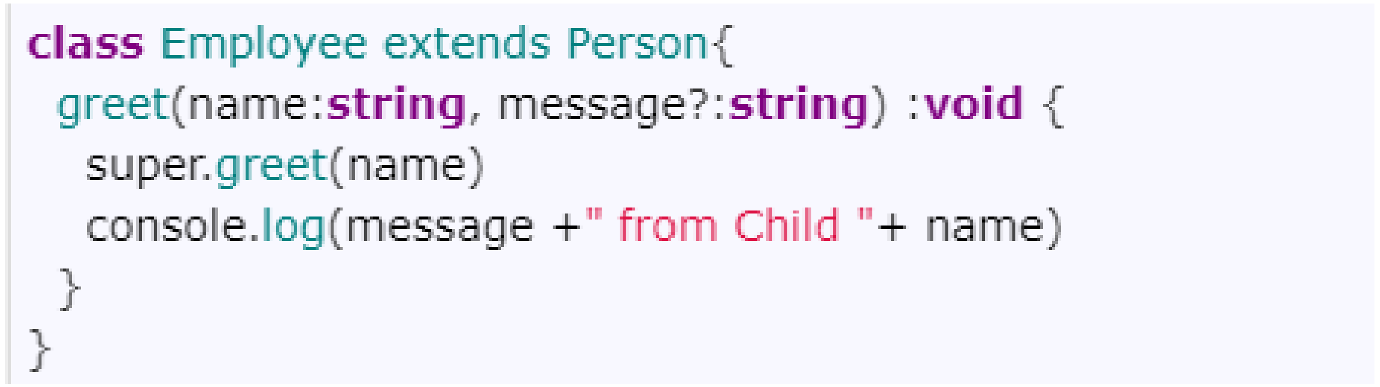
Method Overriding



```
class Person {
  greet(name:string) :void {
    console.log("Hello from Parent "+ name)
  }
}


class Employee extends Person{
  greet(name:string, message:string) :void {
    super.greet(name)
    console.log(message +" from Child "+ name)
  }
}


//Property 'greet' in type 'Employee' is not assignable to the same property in base type 'Person'.
//  Type '(name: string, message: string) => void' is not assignable to type '(name: string) => void'.
```

Method Overriding

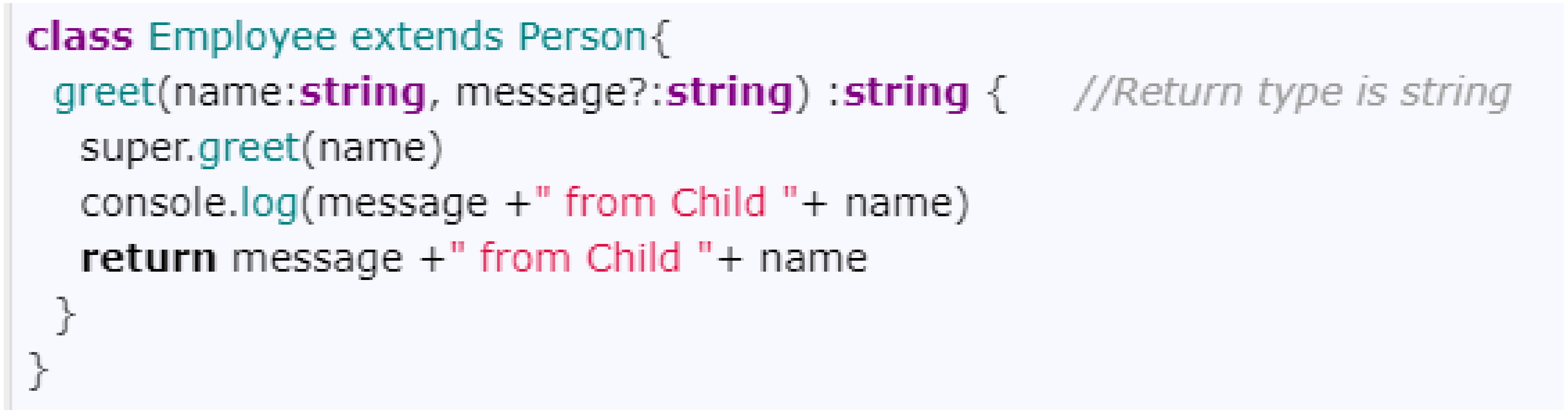
- But you can make message as optional.



```
class Employee extends Person{
  greet(name:string, message?:string) :void {
    super.greet(name)
    console.log(message +" from Child "+ name)
  }
}
```

Method Overriding

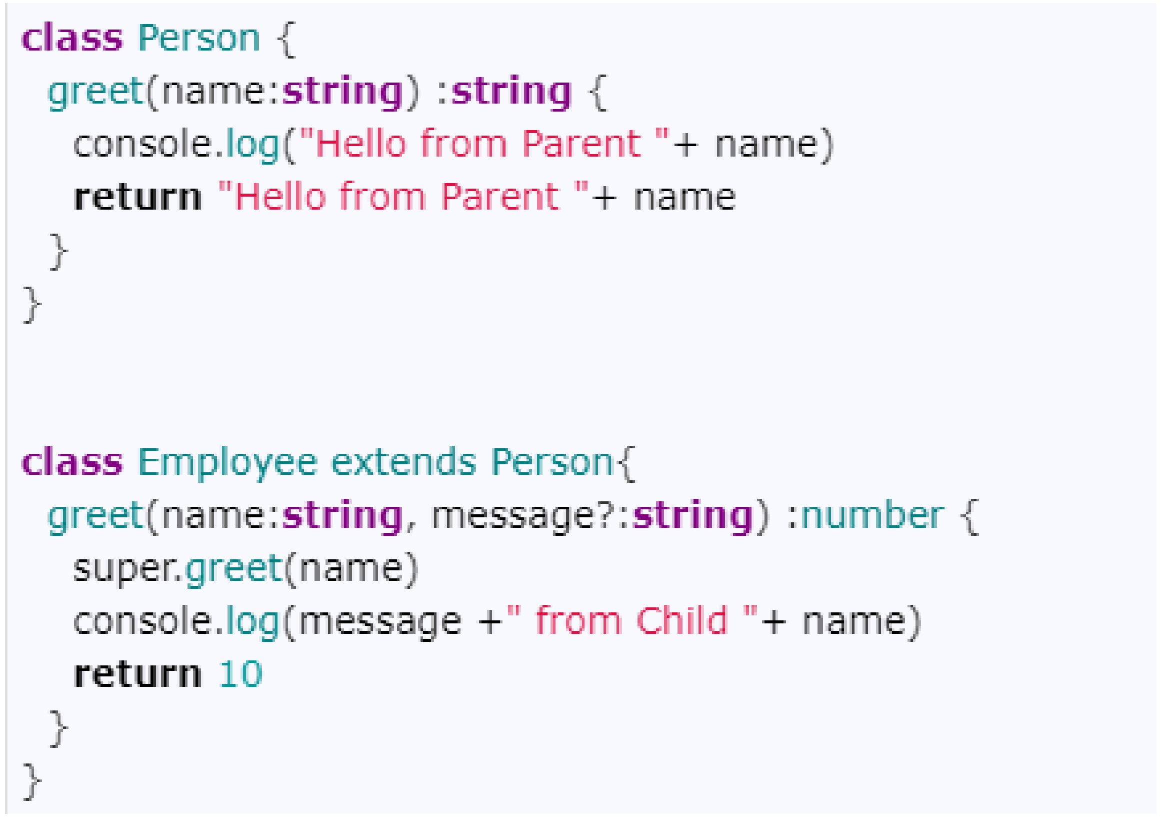
- You can also return string from the child greet method. The parent greet method returns void. Although the void is not compatible with the string compiler does not raise any errors. This is because If a function specifies a return type as void **can return *any* other value**, the compiler will ignore it.



```
class Employee extends Person{
  greet(name:string, message?:string) :string {    //Return type is string
    super.greet(name)
    console.log(message +" from Child "+ name)
    return message +" from Child "+ name
  }
}
```

Method Overriding

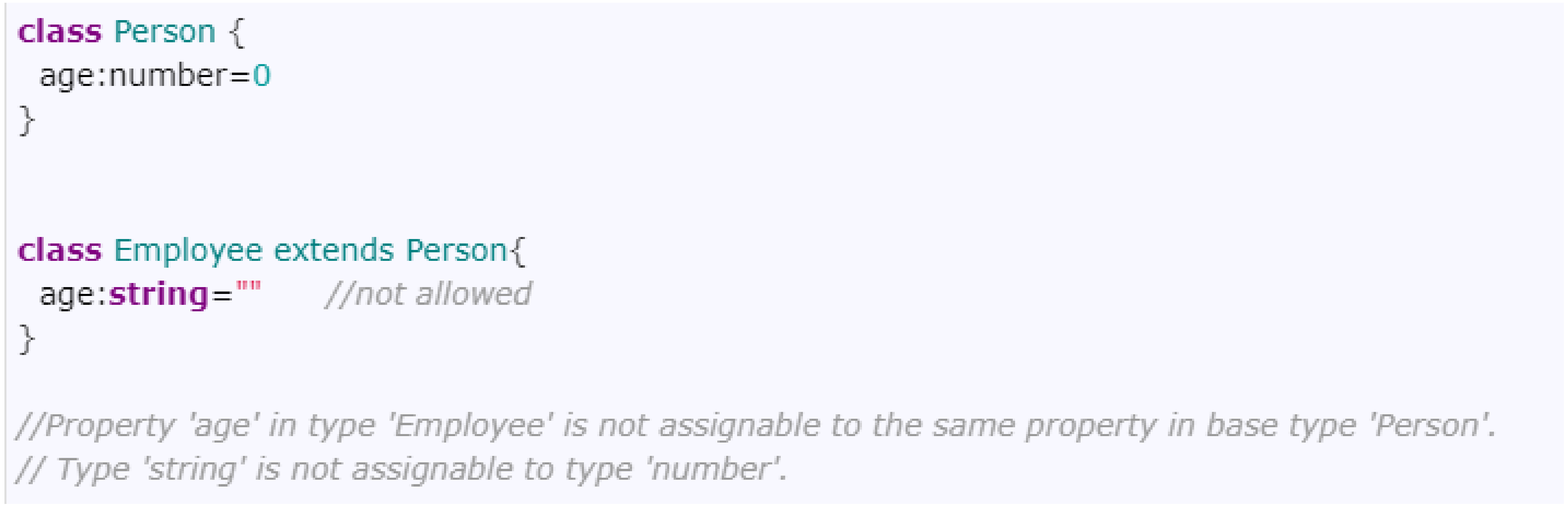
- But you cannot use incompatible return types. The greet method in the Parent class returns a string while the child returns a number. Since these two types are incompatible the compiler throws an error.



```
class Person {
  greet(name:string) :string {
    console.log("Hello from Parent "+ name)
    return "Hello from Parent "+ name
  }
}


class Employee extends Person{
  greet(name:string, message?:string) :number {
    super.greet(name)
    console.log(message +" from Child "+ name)
    return 10
  }
}
```

Property Overriding

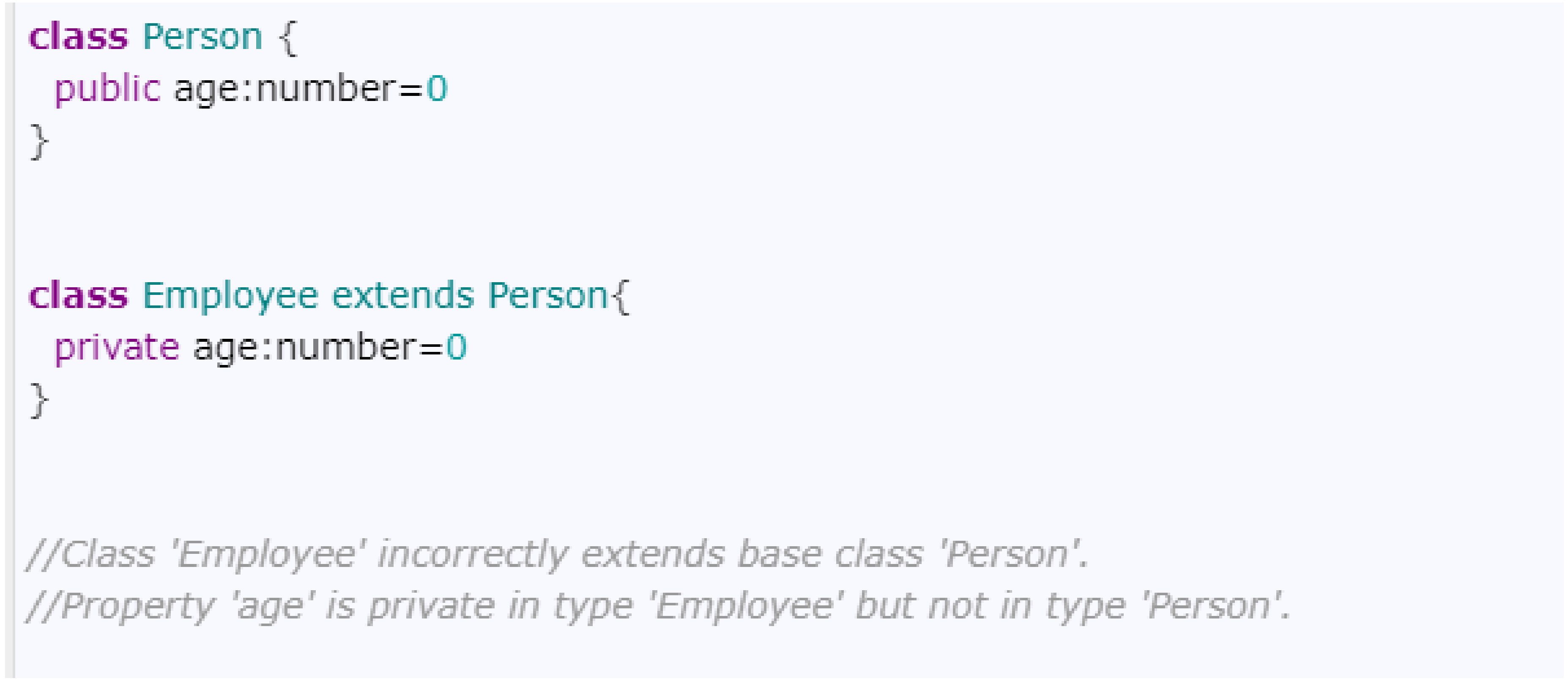
- Similarly, you can override a property, as long as the data types are compatible.



```
class Person {
  age:number=0
}


class Employee extends Person{
  age:string=""      //not allowed
}

//Property 'age' in type 'Employee' is not assignable to the same property in base type 'Person'.
// Type 'string' is not assignable to type 'number'.
```

Property Overriding

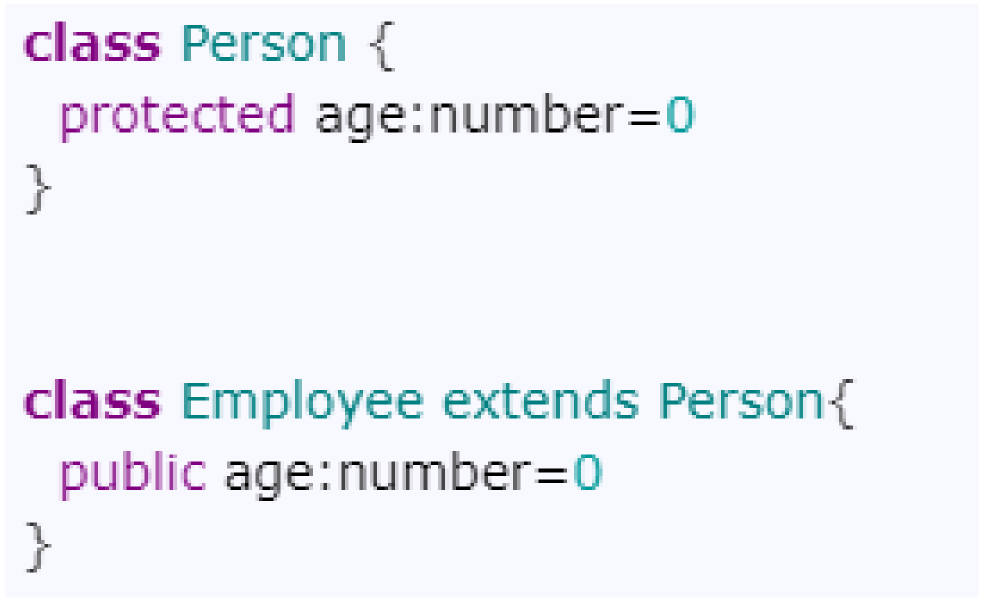
- Typescript does not allow us to change the visibility of the properties in the overridden class **except for changing <u>protected</u> to <u>public</u>**.



```
class Person {
  public age:number=0
}


class Employee extends Person{
  private age:number=0
}


//Class 'Employee' incorrectly extends base class 'Person'.
//Property 'age' is private in type 'Employee' but not in type 'Person'.
```

Property Overriding

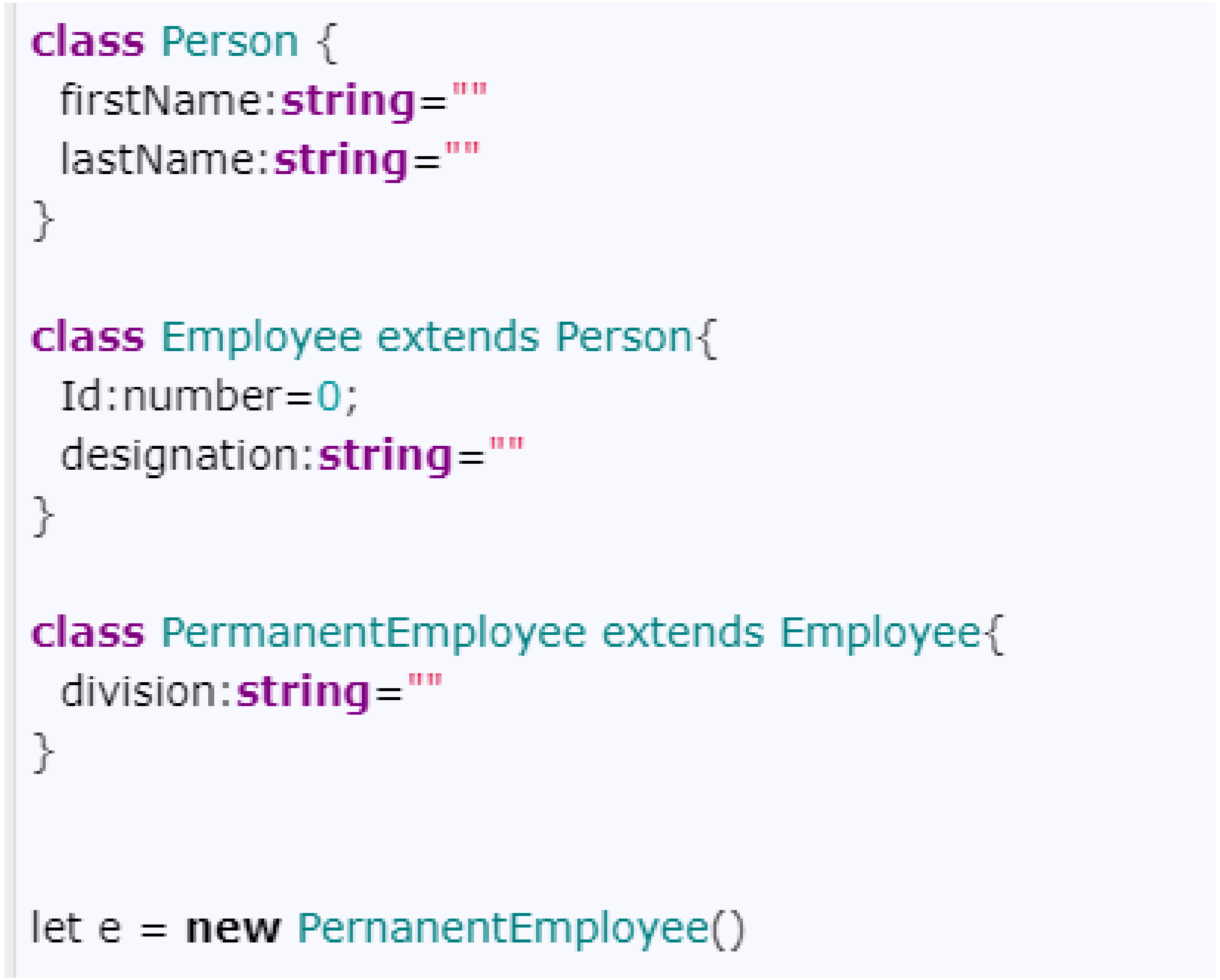
- But changing from protected to public is allowed.



```
class Person {
  protected age:number=0
}


class Employee extends Person{
  public age:number=0
}
```

Multi-level Inheritance

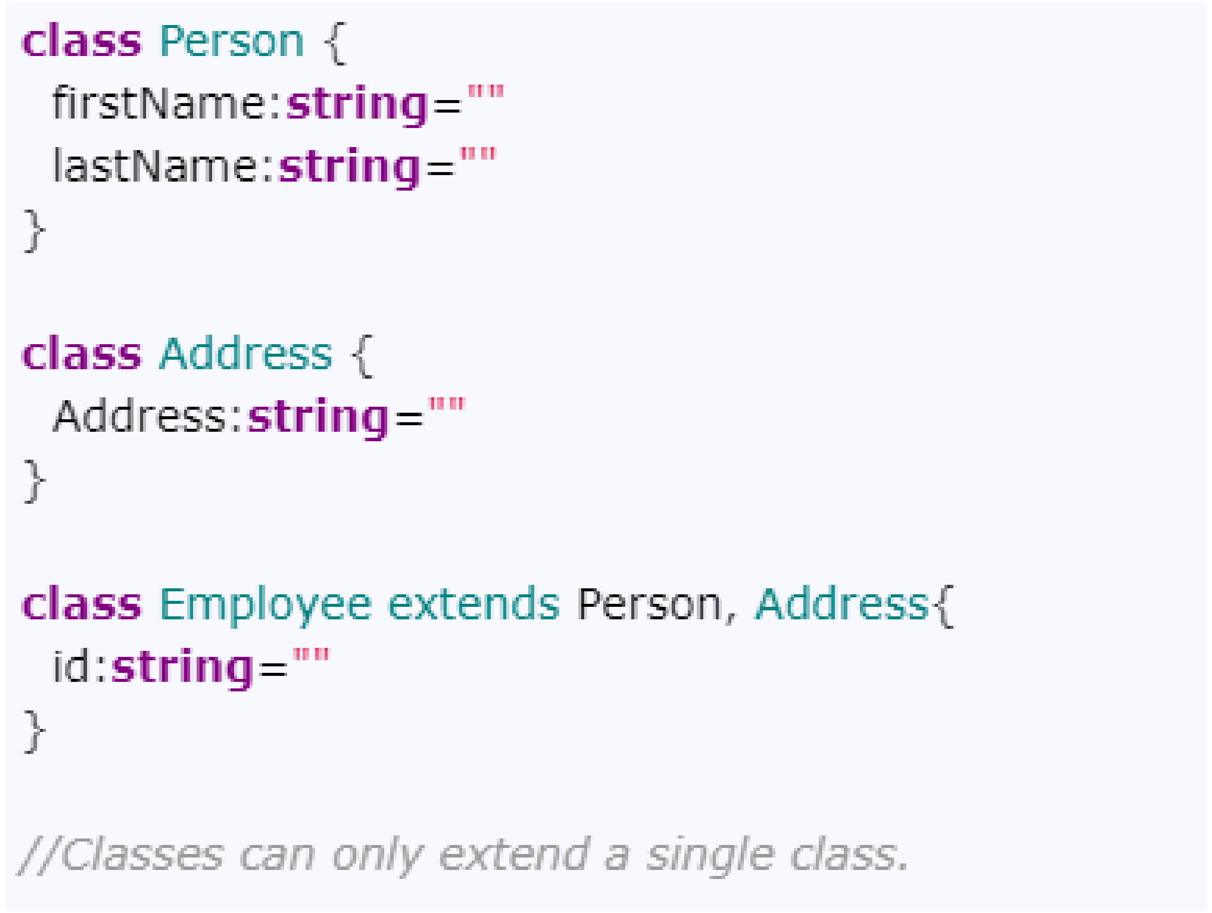
- Inheritance in typescript can go up to any level. The following is an example of three-level of inheritance.



```
class Person {
  firstName:string=""
  lastName:string=""
}

class Employee extends Person{
  Id:number=0;
  designation:string=""
}

class PermanentEmployee extends Employee{
  division:string=""
}


let e = new PernanentEmployee()
```

Multi-level Inheritance

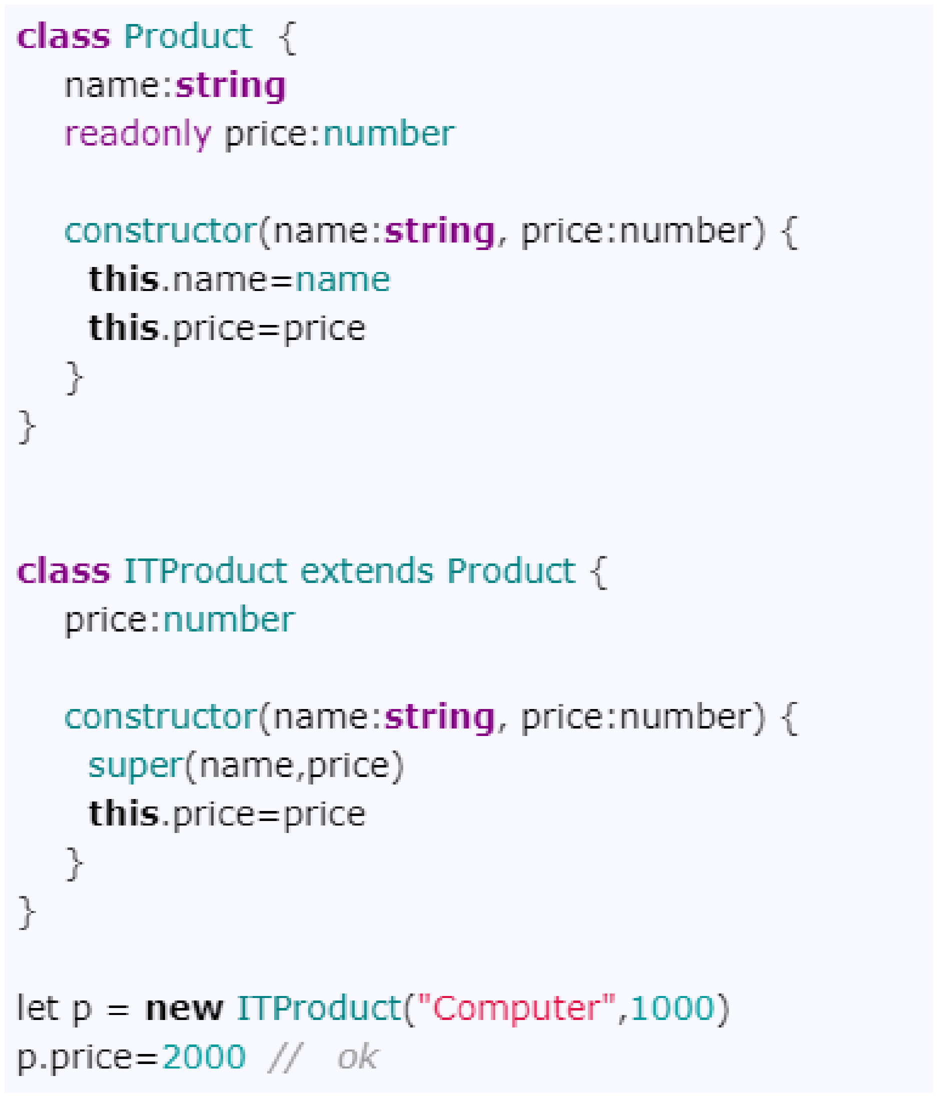
- Typescript does not allow us to inherit from more than one class.



```
class Person {
  firstName:string=""
  lastName:string=""
}

class Address {
  Address:string=""
}

class Employee extends Person, Address{
  id:string=""
}

//Classes can only extend a single class.
```

Overriding Readonly & Optional Properties

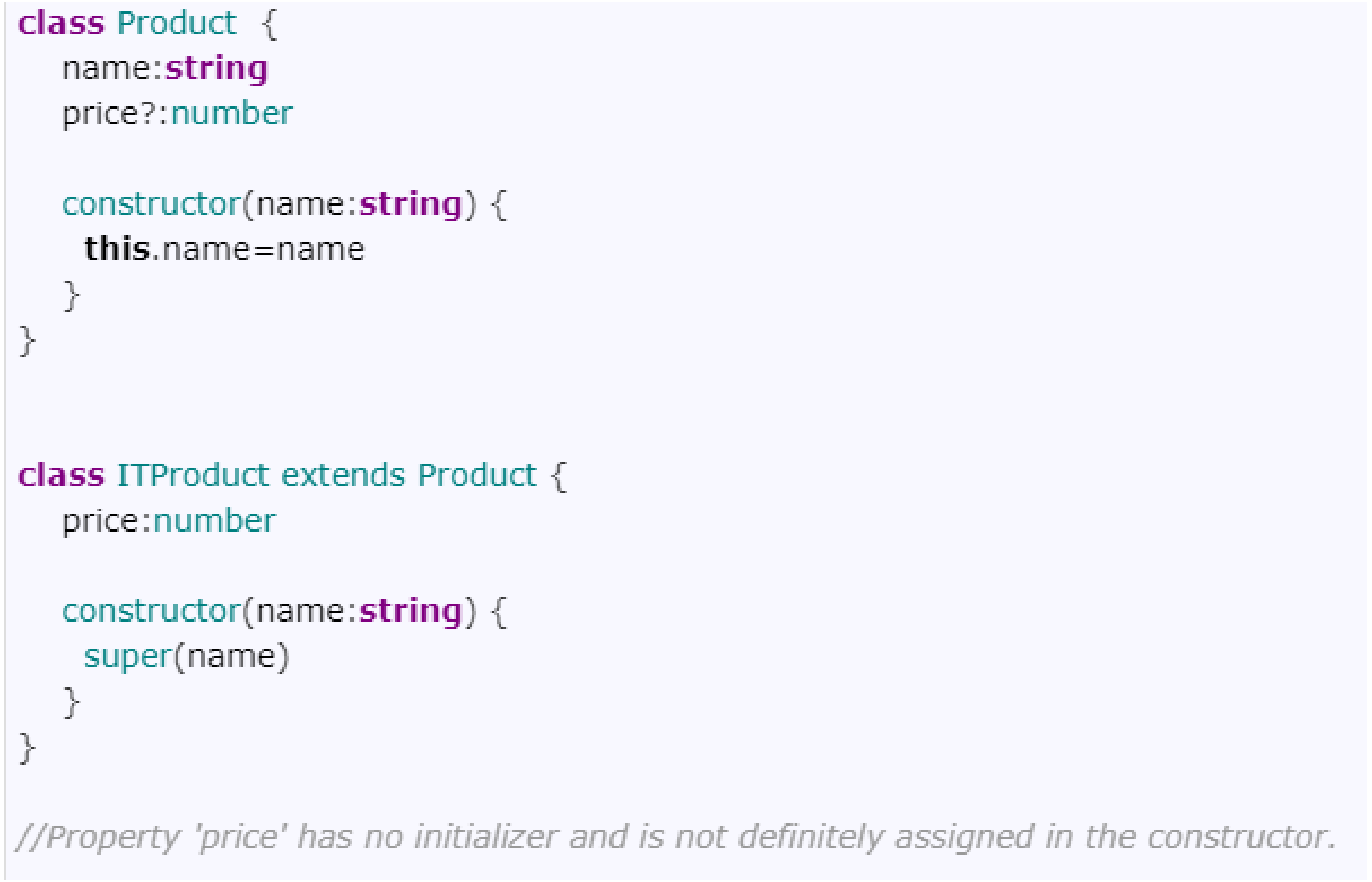
- The derived classes can override the Readonly and Optional properties of Public and Protected Properties. It can either make the property Readonly and /or Optional or Remove it



```
class Product  {
   name:string
   readonly price:number

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


class ITProduct extends Product {
   price:number

   constructor(name:string, price:number) {
     super(name,price)
     this.price=price
   }
}

let p = new ITProduct("Computer",1000)
p.price=2000 //  ok
```

Overriding Readonly & Optional Properties

- Similarly, the derived class removes the optional from the price property.



```
class Product  {
  name:string
  price?:number

  constructor(name:string) {
   this.name=name
  }
}


class ITProduct extends Product {
  price:number

  constructor(name:string) {
   super(name)
  }
}

//Property 'price' has no initializer and is not definitely assigned in the constructor.
```

Getters and Setters in TypeScript

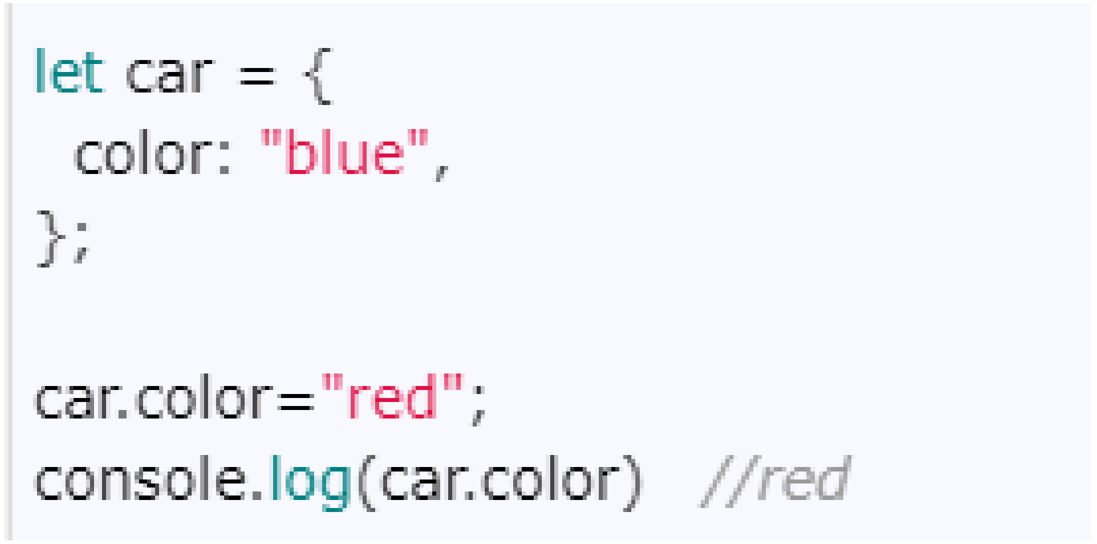
- The Getters and Setters are known as accessor properties in TypeScript.
- They look like normal properties but are actually functions mapped to a Property.
- We use “**get**” to define a **getter** method and “**set**” to define a **setter method**.
- The Setter method runs when we assign a value to the Property. The Getter method runs when we access the Property.

What are Getters & Setters

- The Getters and Setters are known as **Accessor Properties** in TypeScript.
- The Properties of an object can be accessed in two ways. We can access **them directly** or **through a function**. Based on how we access them, the properties are classified as
  1. Data Properties
  2. Accessor Properties
- The **Data Property** is mapped to a value. The value can be a primitive value, object, or function.

Data Property

- In the example below, the property color is data property. We can access its value as car.color or assign a new value to it using car.color="red"



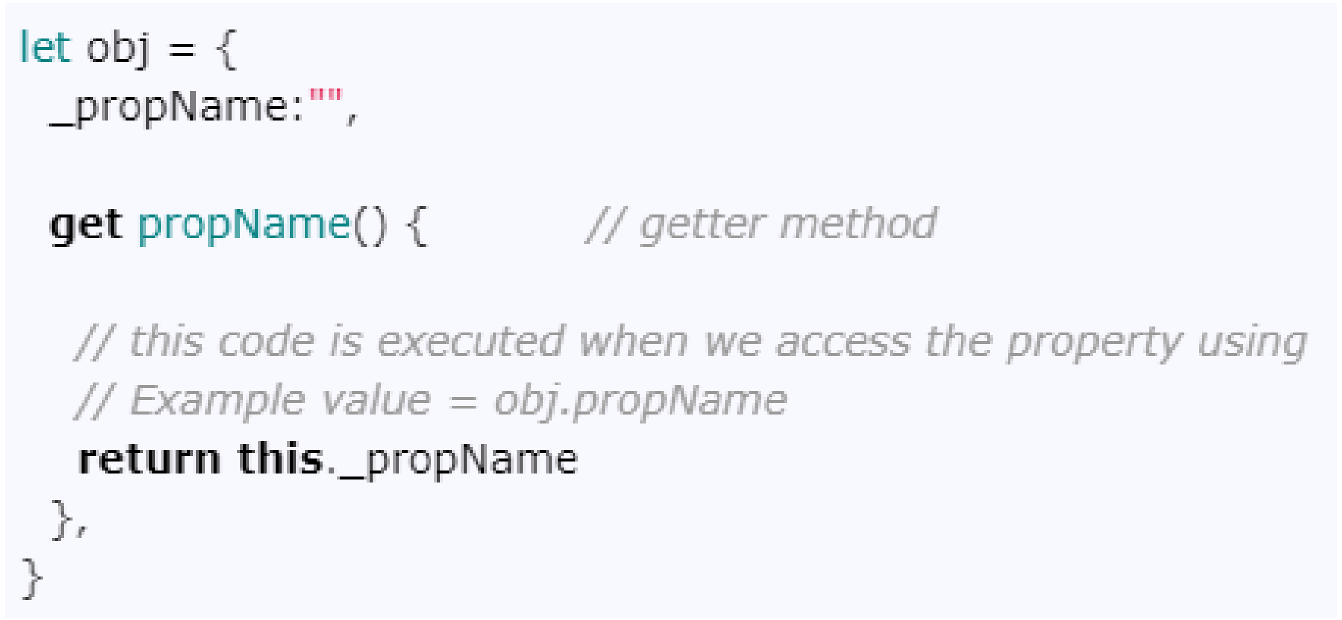
```
let car = {
  color: "blue",
};

car.color="red";
console.log(car.color)  //red
```

Accessor property

- The **accessor property** is not mapped to a value but to a **function**. We call this function as **accessor function**. It is the job of the function to store or retrieve the value.
- The accessor function that **retrieves the value of a property is known as the Getter method**. We use the **Get keyword** to declare a Getter method.
- The accessor function that **assigns a value to a property is known as the Setter method**. We use the **Set keyword** to declare a Setter method.

Creating Getters TypeScript

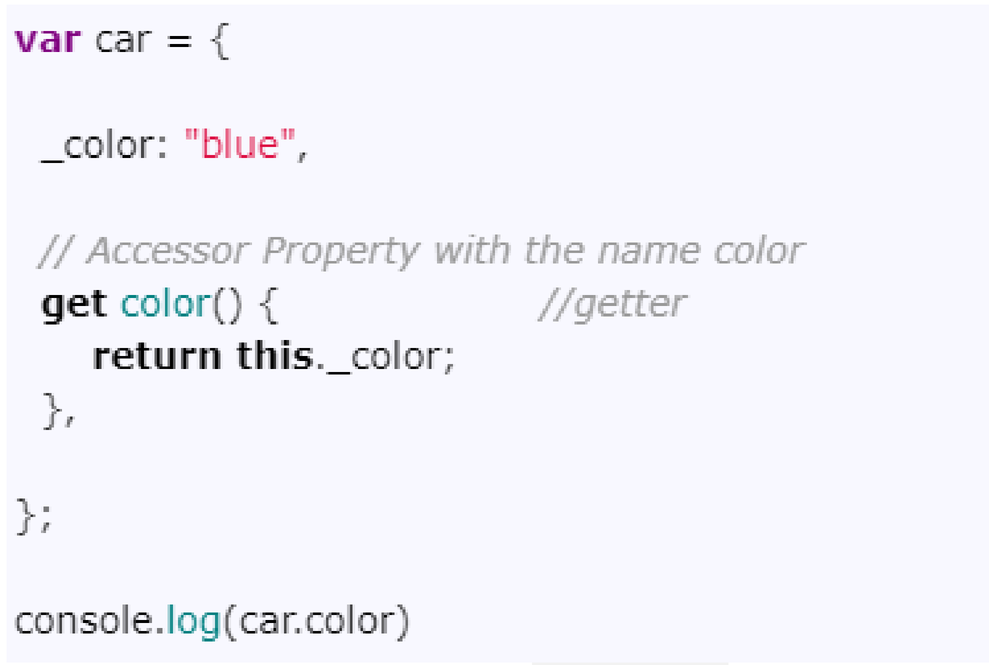
- We use the **get** keyword followed by a function expression. The name of the function becomes the name of the property (**propName**).



```
let obj = {
  _propName:"",

  get propName() {          // getter method

    // this code is executed when we access the property using
    // Example value = obj.propName
    return this._propName
  },
}
```

- The getter function executes when we read the value of the Property. Note that **we cannot pass an argument to the getter method**. The return value of the getter method becomes the value of the property access expression.

Creating Getters TypeScript

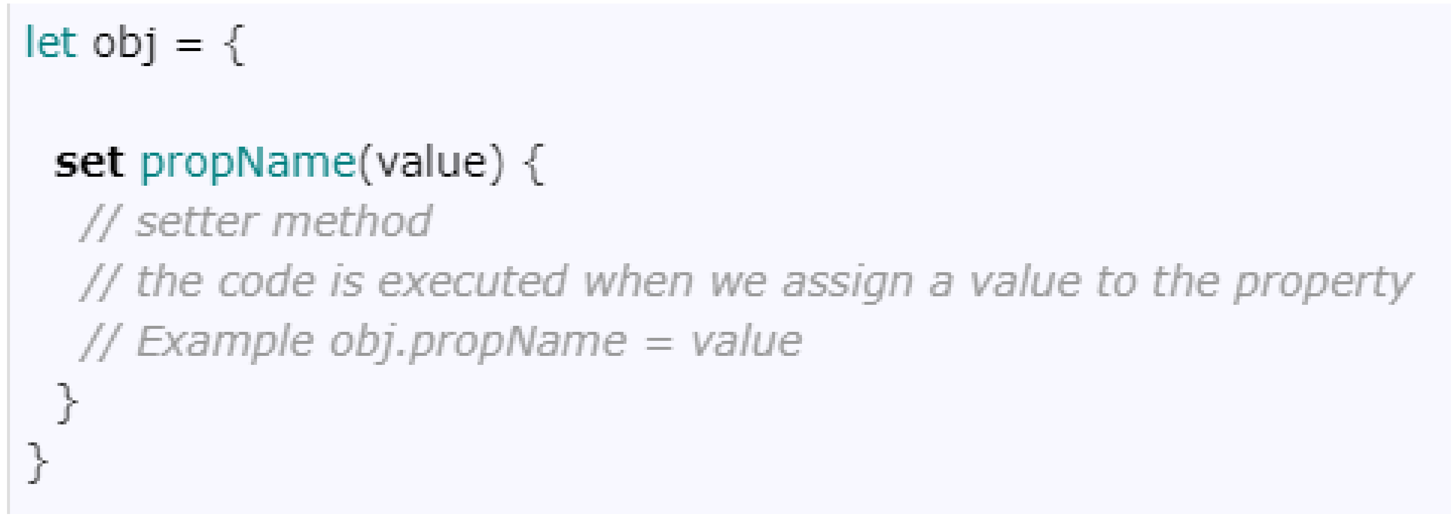


```
var car = {

  _color: "blue",

  // Accessor Property with the name color
  get color() {                  //getter
     return this._color;
  },

};

console.log(car.color)
```

- We access the color getter property just like any other Javascript Property. i.e. using the dot notation.
- Note that although the color is a function, we do not invoke it like a function i.e car.color() But access it just like a property i.e. car.color

Creating Setters TypeScript

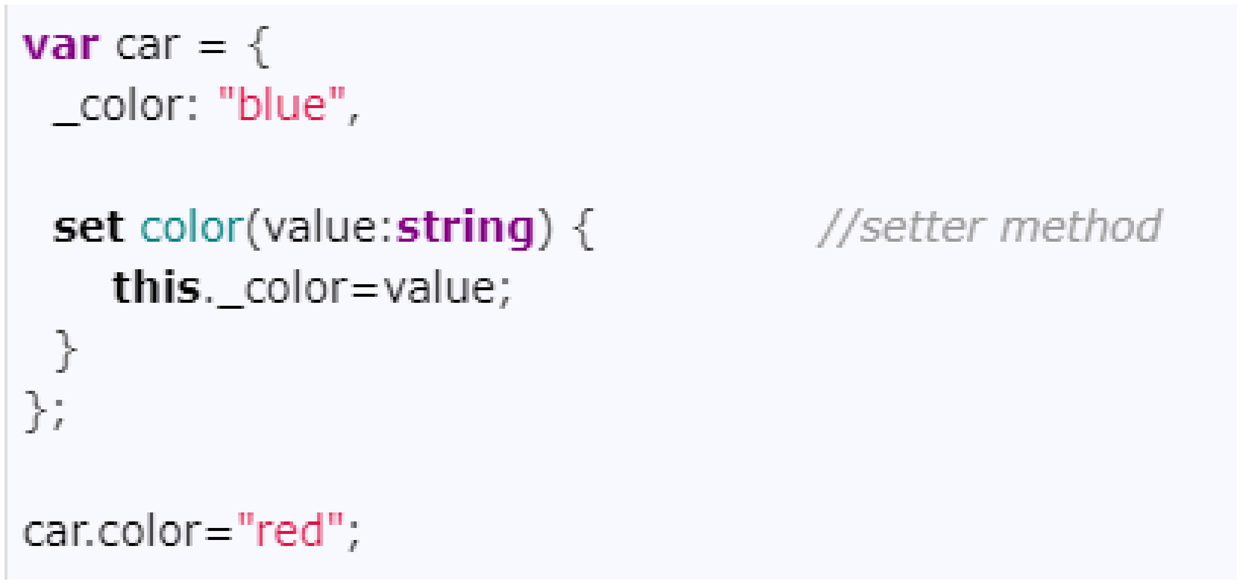
- To create a setter method, we use the **set** keyword followed by a function expression. The name of the function becomes the name of the property (**propName**). **A 'set' accessor must have exactly one parameter**.



```
let obj = {

  set propName(value) {
    // setter method
    // the code is executed when we assign a value to the property
    // Example obj.propName = value
  }
}
```

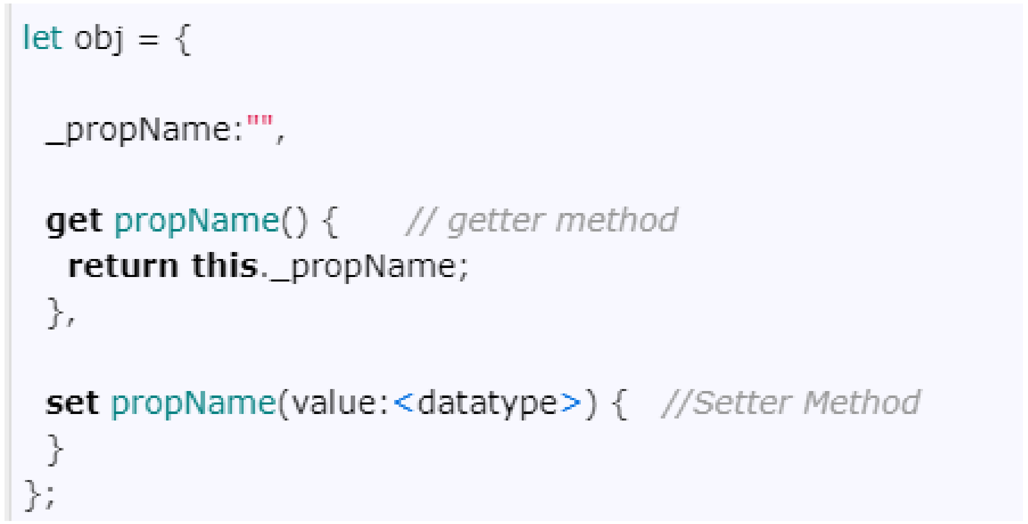
- The **setter** method executes when we assign a value to the property. JavaScript invokes the setter method with the value of the right-hand side of the assignment as the argument.

Creating Setters TypeScript

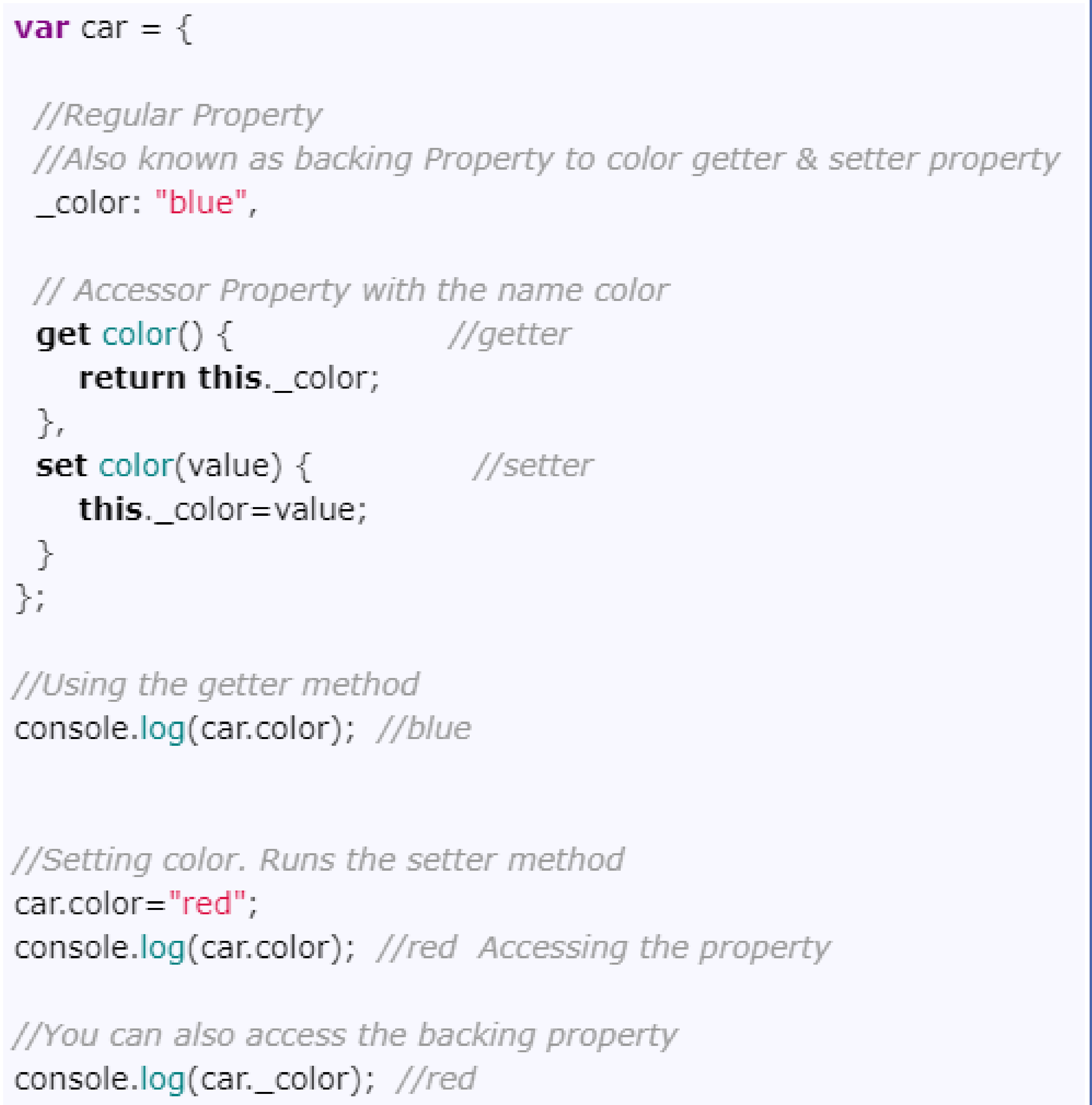


```
var car = {
  _color: "blue",

  set color(value:string) {          //setter method
     this._color=value;
  }
};

car.color="red";
```

Getter & Setters



```
let obj = {

  _propName:"",

  get propName() {      // getter method
    return this._propName;
  },

  set propName(value:<datatype>) {   //Setter Method
  }
};
```



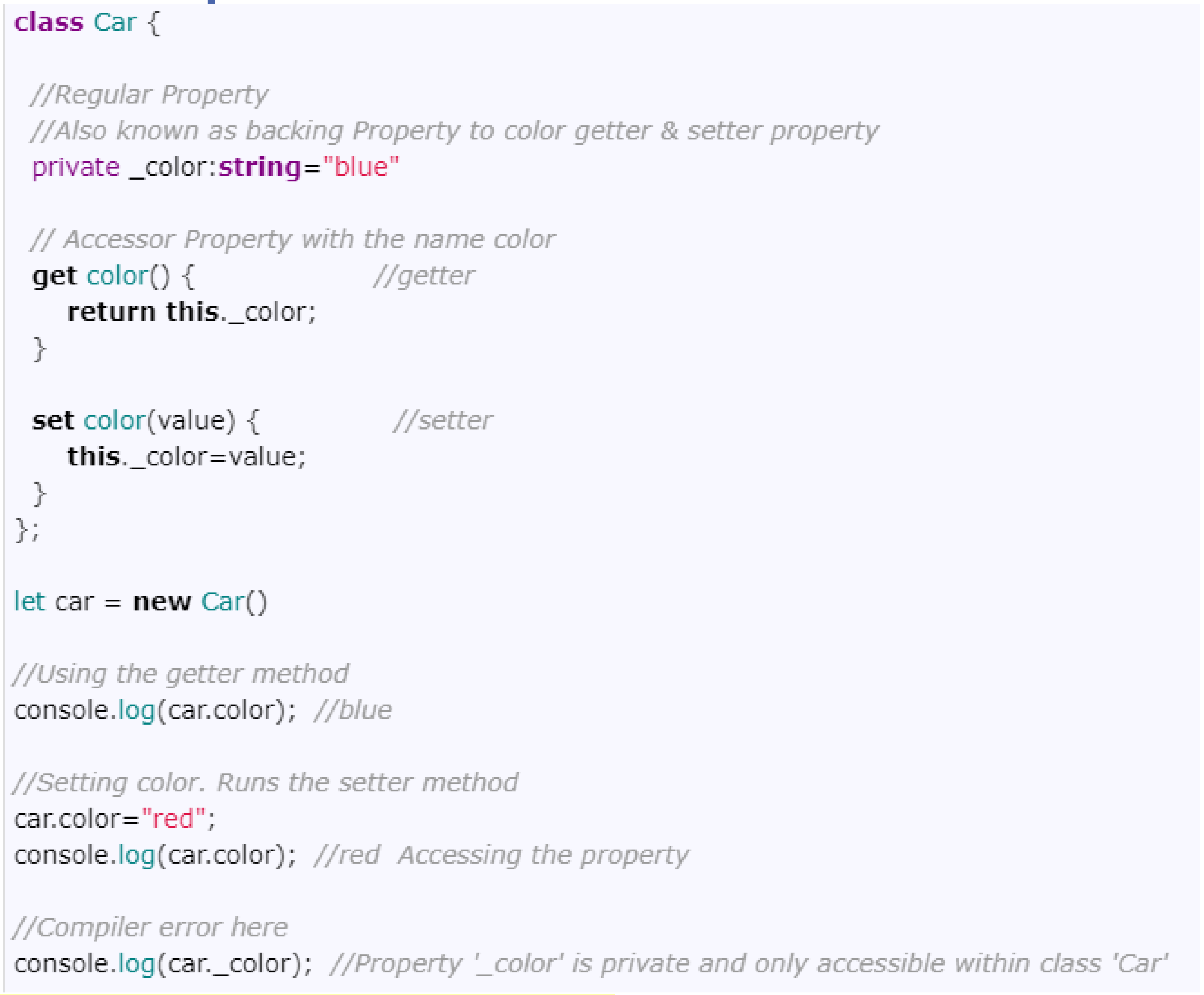
```
var car = {

  //Regular Property
  //Also known as backing Property to color getter & setter property
  _color: "blue",

  // Accessor Property with the name color
  get color() {                    //getter
      return this._color;
  },
  set color(value) {                //setter
      this._color=value;
  }
};

//Using the getter method
console.log(car.color);  //blue


//Setting color. Runs the setter method
car.color="red";
console.log(car.color);  //red  Accessing the property

//You can also access the backing property
console.log(car._color);  //red
```

Getter & Setters

- Notice that the color accessor property behind the scene uses the _color property to store the value. _color property is the **backing property of the Color accessor property**. we prepend the backing property with an underscore to indicate that _color should not be accessed directly.
- Note that you cannot use the same name for an accessor property and a regular property.

Getter & Setters in TypeScript Classes

- We can create getters and setters in TypeScript classes, in the same way, we create them in TypeScript objects.
- In the Typescript Classes, we can use the private access Modifier and mark the backing property as private.
- Marking the backing property _color as private ensures that we do not accidentally modify its value directly. The Compiler throws the error

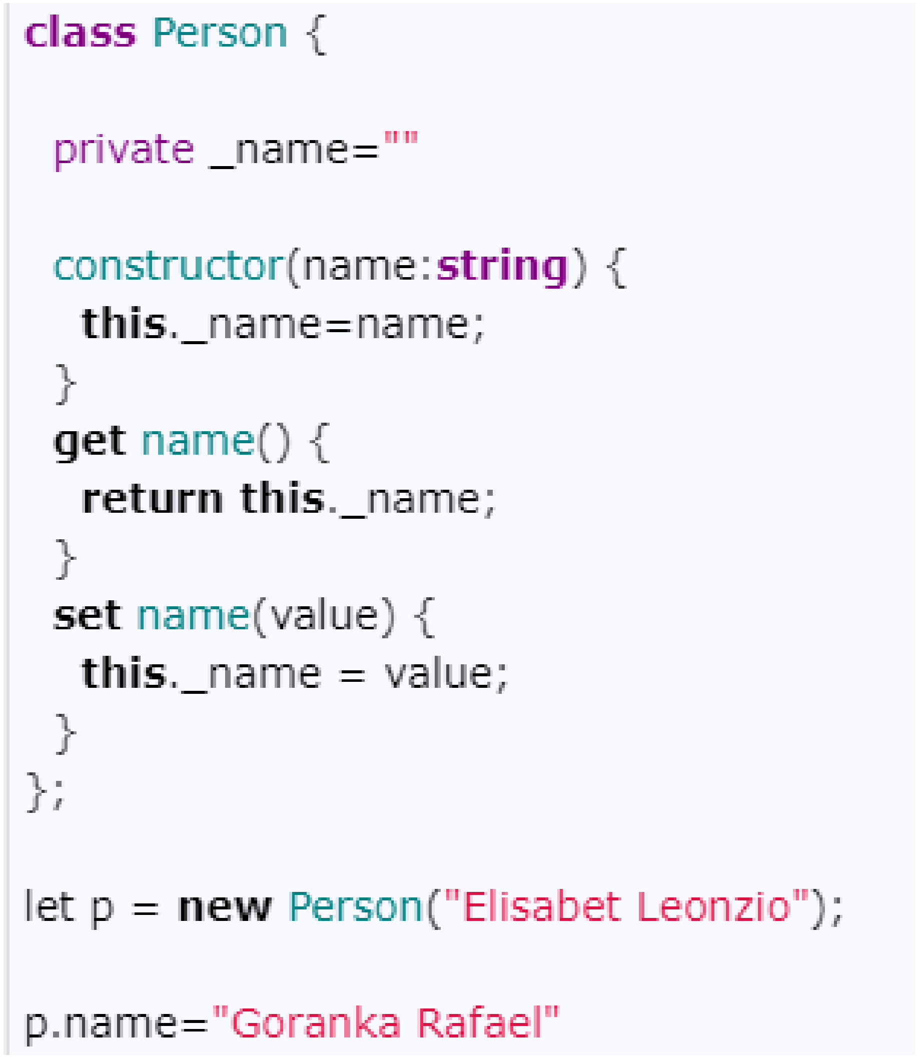


```
class Car {

  //Regular Property
  //Also known as backing Property to color getter & setter property
  private _color:string="blue"

  // Accessor Property with the name color
  get color() {                    //getter
     return this._color;
  }

  set color(value) {               //setter
     this._color=value;
  }
};

let car = new Car()

//Using the getter method
console.log(car.color);  //blue

//Setting color. Runs the setter method
car.color="red";
console.log(car.color);  //red  Accessing the property

//Compiler error here
console.log(car._color);  //Property '_color' is private and only accessible within class 'Car'
```

Property <propertyName> is private and only accessible within class <className>

Getter & Setters in TypeScript Classes

- There is no need to use accessor methods if you are simply using them to get or set the data property as in the example below. Plain property access is a better option here.



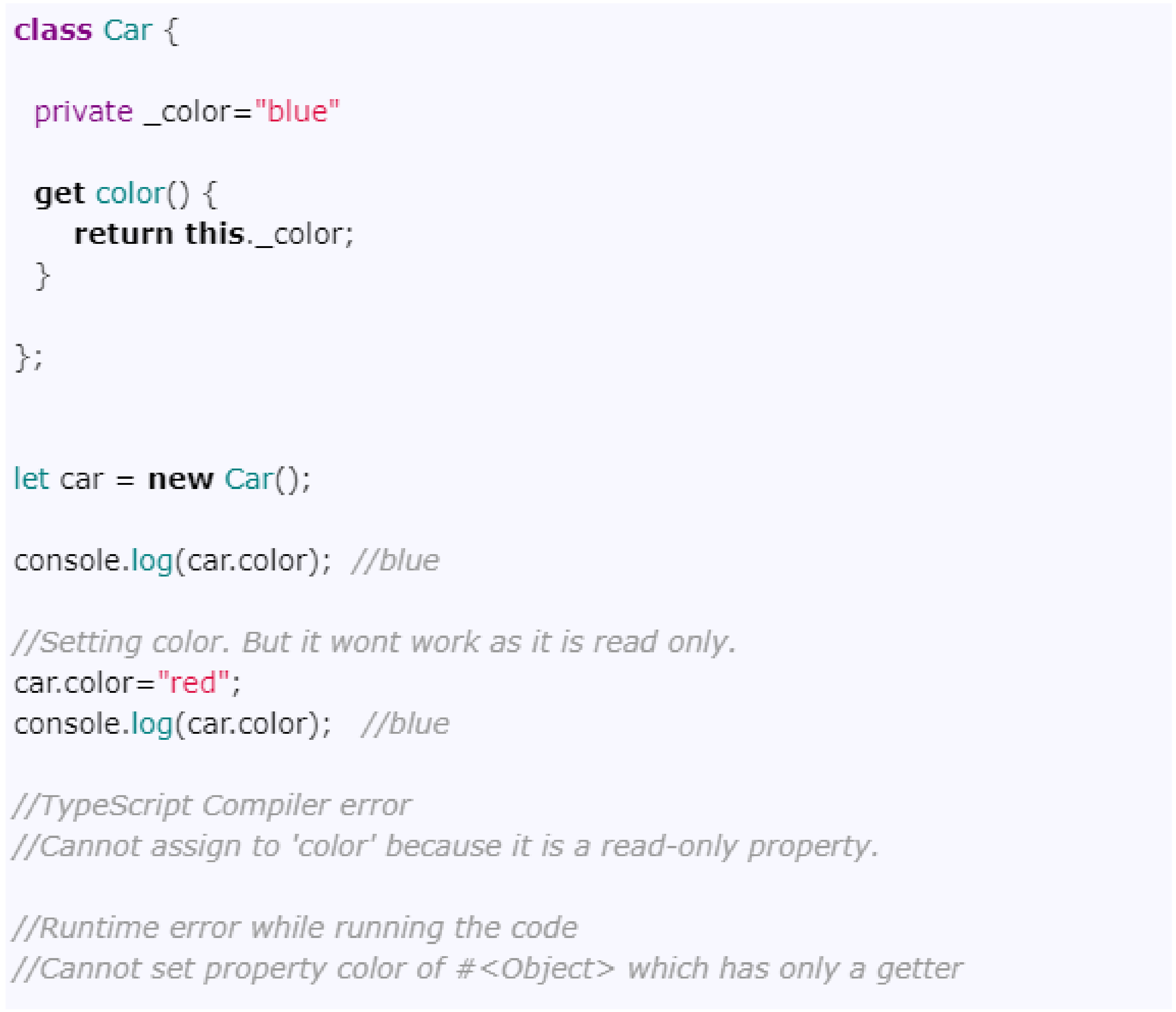
```
class Person {

 private _name=""

 constructor(name:string) {
  this._name=name;
 }
 get name() {
  return this._name;
 }
 set name(value) {
  this._name = value;
 }
};

let p = new Person("Elisabet Leonzio");

p.name="Goranka Rafael"
```

Read-only / Write-only Properties

- We can use the Setters & getters to create read-only or write-only properties.
- **If the property has only a getter method, then it is a read-only property. If it has only a setter method then it is a write-only property**

Read-only / Write-only Properties

- In this example, we only define a get method, making the color property read-only. Setting the color property using an assignment (car.color="red";) will result in a Compiler error and also an error while running the code



```
class Car {

  private _color="blue"

  get color() {
     return this._color;
  }

};


let car = new Car();

console.log(car.color);  //blue

//Setting color. But it wont work as it is read only.
car.color="red";
console.log(car.color);   //blue

//TypeScript Compiler error
//Cannot assign to 'color' because it is a read-only property.

//Runtime error while running the code
//Cannot set property color of #<Object> which has only a getter
```