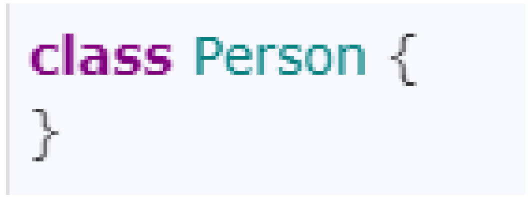
Typescript TS

Classses

Classes in TypeScript

- Classes in TypeScript is a template **for creating objects**.
- The class **defines the data** (fields or properties) and **the operations** (methods) that we **can perform on that data**.
- We can create **many objects** using a **single class**.
- To create a class we need to use the **class** keyword followed by **{ }**. Inside the brackets, we will **add the properties**, **methods**, and a **special constructor method**.



```
class Person {
}
```

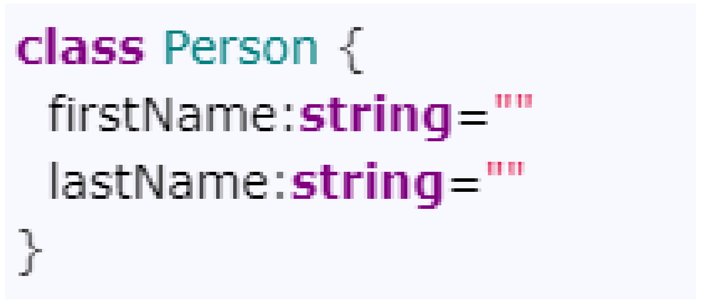
Classes in TypeScript

A class declaration can contain the following

1. **Fields**: A field (or property) is a variable that we declare in the class. **It will contain the data of the class**.
2. **Functions or Methods**: The Operations that the objects can perform. **They represent the behavior of the class**.
3. **Constructor**: is a **special function** that is invoked every time we **create a new object from the class**.

Adding Fields to Class

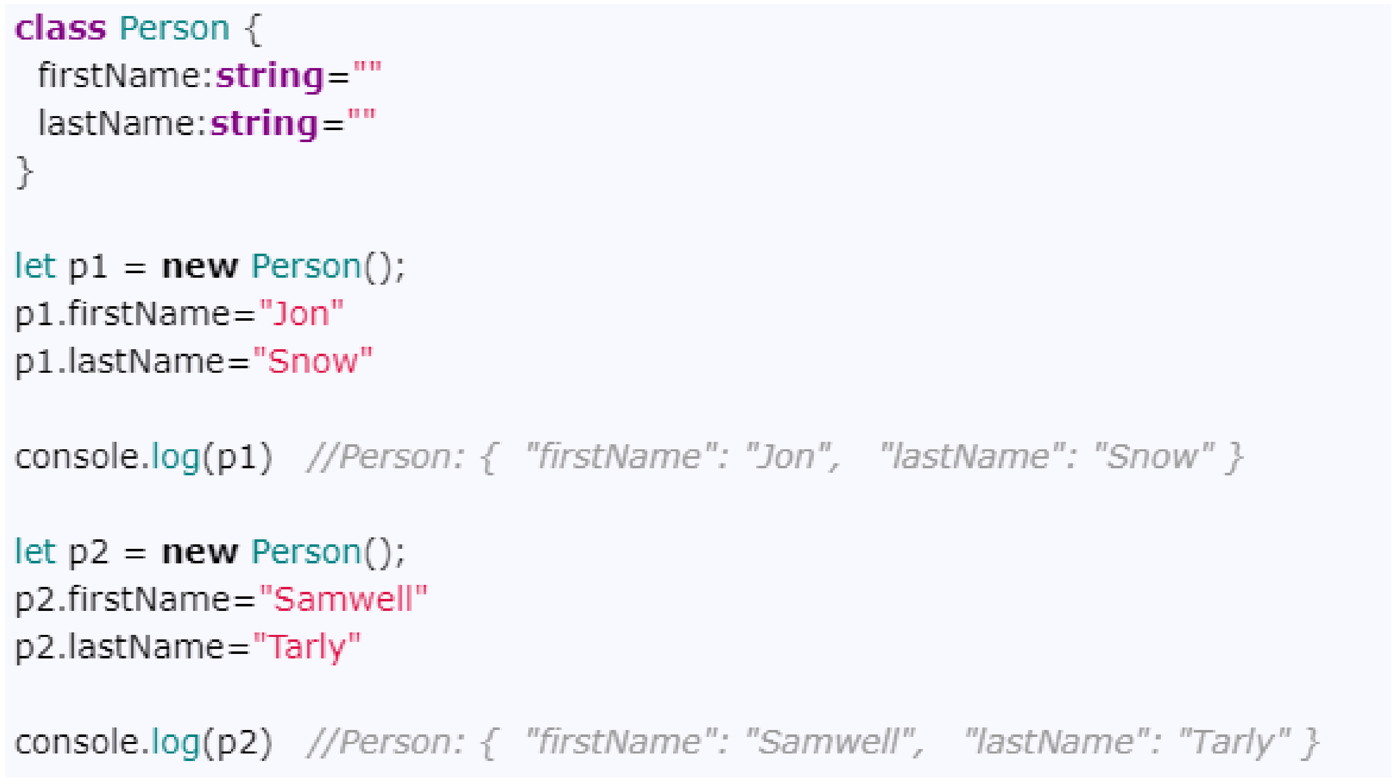
- The fields (properties) hold the data of the class. **Adding a field is similar to declaring the variables.** But without the use of **let, var** & **const** keyword.



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

Creating objects from Class

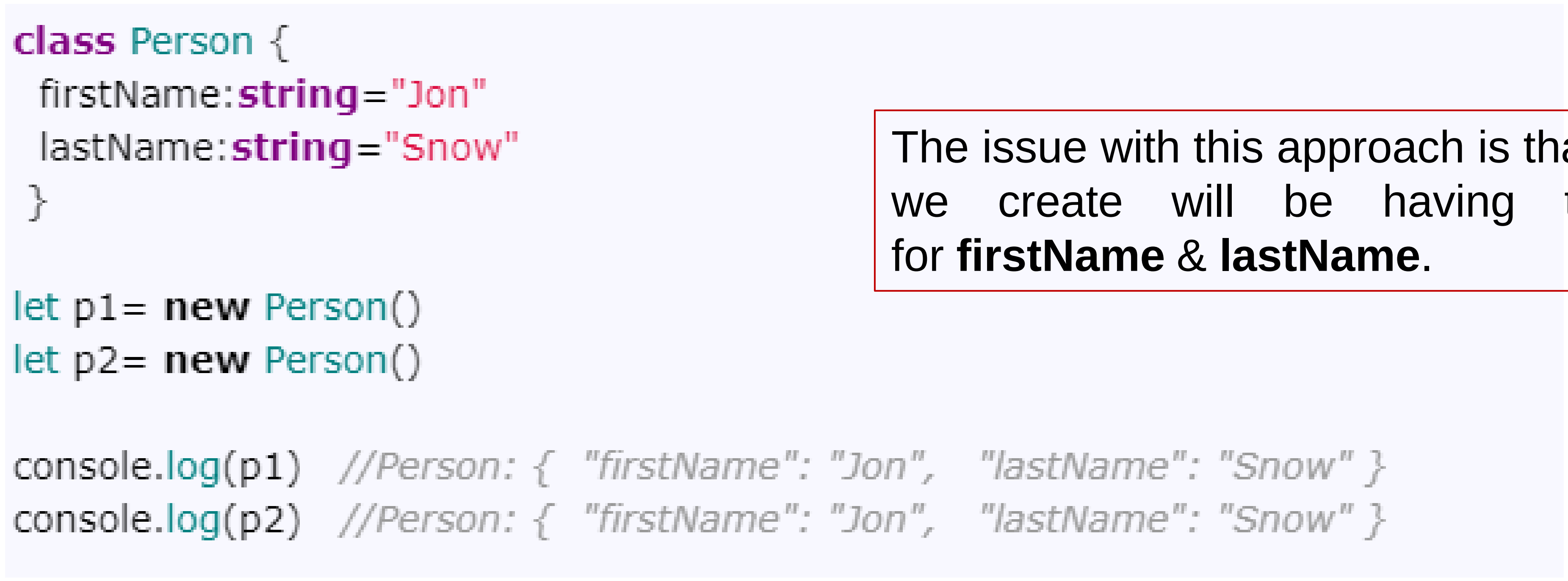
- The class acts as a **template for creating objects**. We can create **multiple objects using the same class**. To create a new object use the **new** keyword followed by the **name of the class** and then brackets **( )**.



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

let p1 = new Person();
p1.firstName="Jon"
p1.lastName="Snow"

console.log(p1)  //Person: {  "firstName": "Jon",   "lastName": "Snow" }

let p2 = new Person();
p2.firstName="Samwell"
p2.lastName="Tarly"

console.log(p2)  //Person: {  "firstName": "Samwell",   "lastName": "Tarly" }
```

Providing initial value to Properties

- There are two ways by which you can provide an initial value to the property. One is **in line with the declaration of property** fields. Second is to use the constructor **function.**

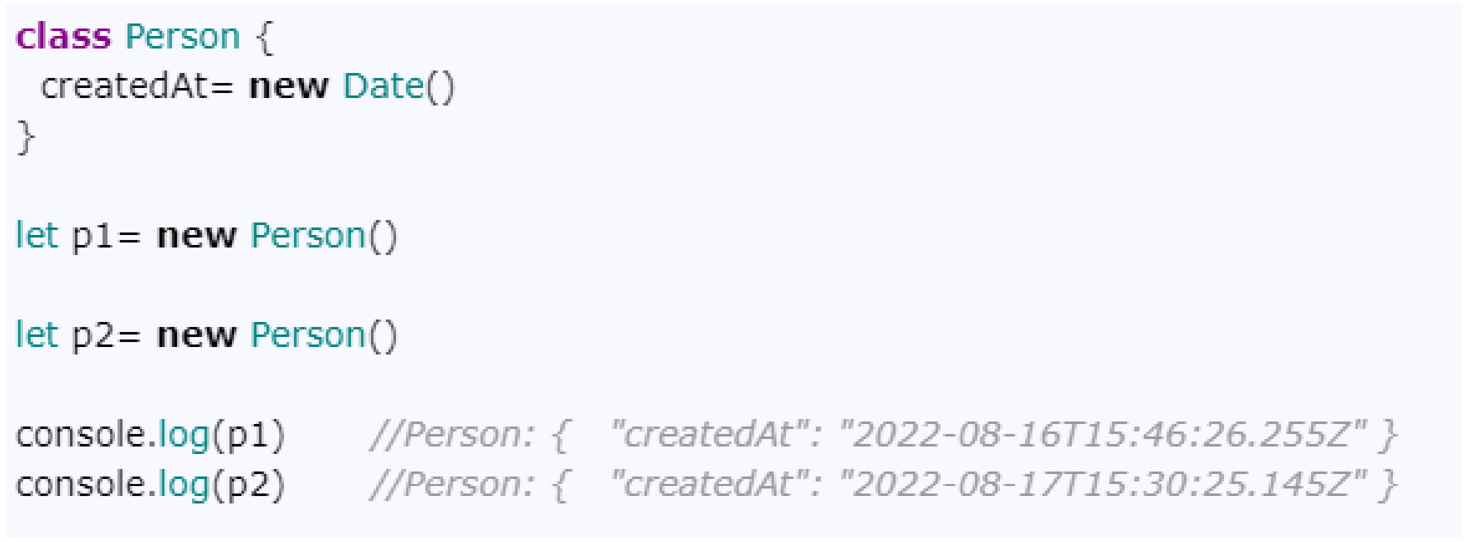


```
class Person {
  firstName:string="Jon"
  lastName:string="Snow"
 }

let p1= new Person()
let p2= new Person()

console.log(p1)  //Person: {  "firstName": "Jon",   "lastName": "Snow" }
console.log(p2)  //Person: {  "firstName": "Jon",   "lastName": "Snow" }
```

The issue with this approach is that every object that we create will be having the same value for **firstName** & **lastName**.

Providing initial value to Properties

- We can also call some functions or methods etc.

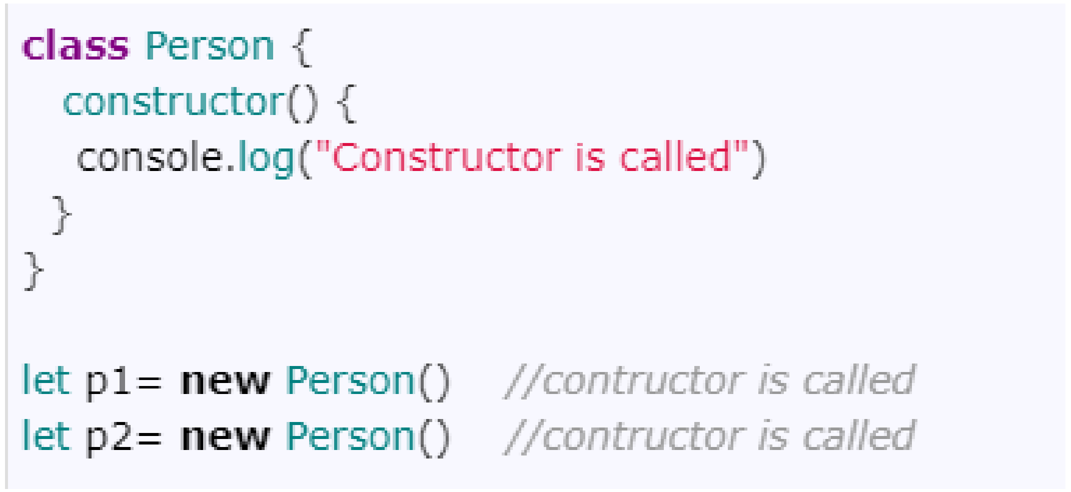


```
class Person {
  createdAt= new Date()
}

let p1= new Person()

let p2= new Person()

console.log(p1)     //Person: {   "createdAt": "2022-08-16T15:46:26.255Z" }
console.log(p2)     //Person: {   "createdAt": "2022-08-17T15:30:25.145Z" }
```

The preferred way is to use the **constructor function**, which allows us to pass different values when we create an object from it.

Constructor function

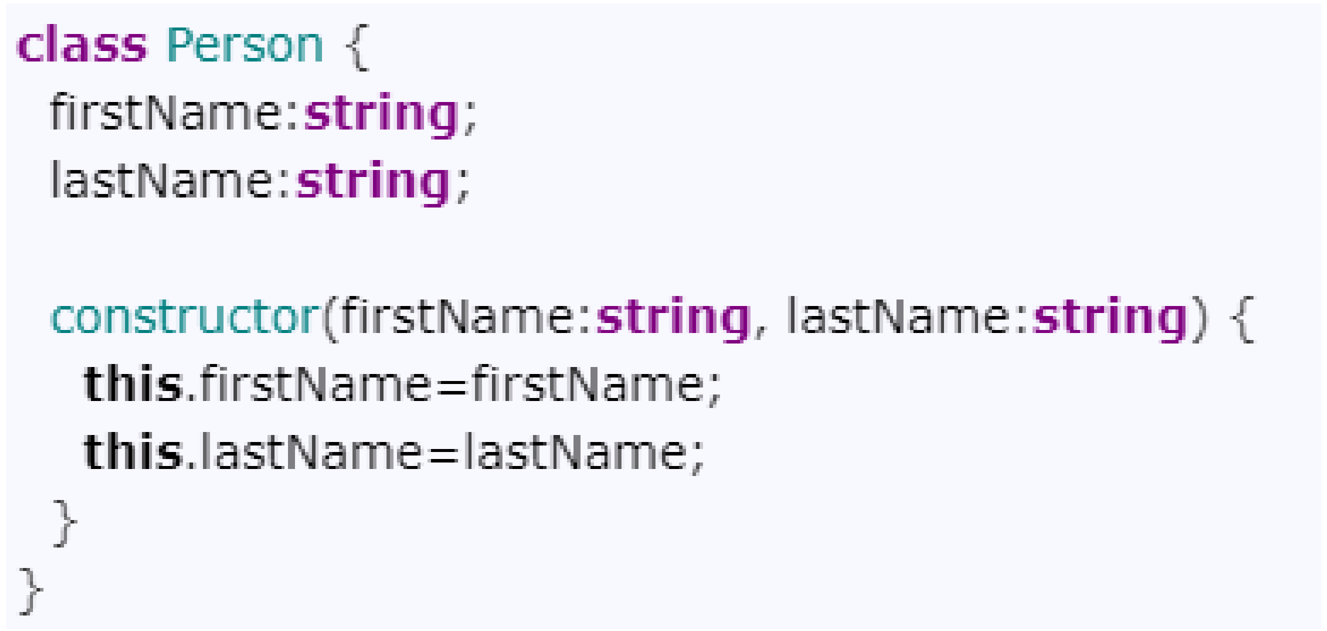
- A constructor is **a special function of the class that is** automatically invoked **when we create an instance of the class**. **We use it to create & initialize the instance of the class**.
- The constructor method in a class must have the name **constructor**. A class can have **only one implementation of the constructor method.**



```
class Person {
  constructor() {
   console.log("Constructor is called")
  }
}

let p1= new Person()   //contructor is called
let p2= new Person()   //contructor is called
```

Constructor function

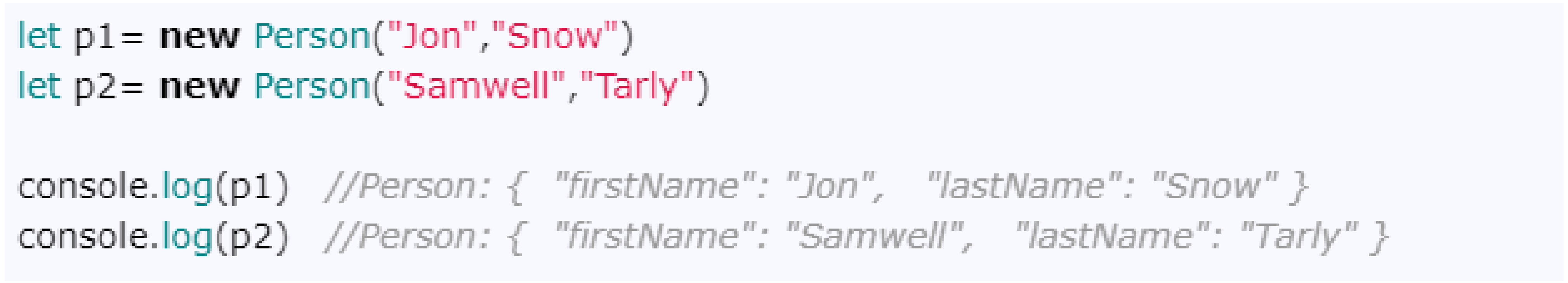
- We use the constructor function to initialize the instance of the class. **The constructor function is just like any other function and can accept any number of arguments.**



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

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

Constructor function

- We can create an **instance from the class by invoking the new operator on the class.**

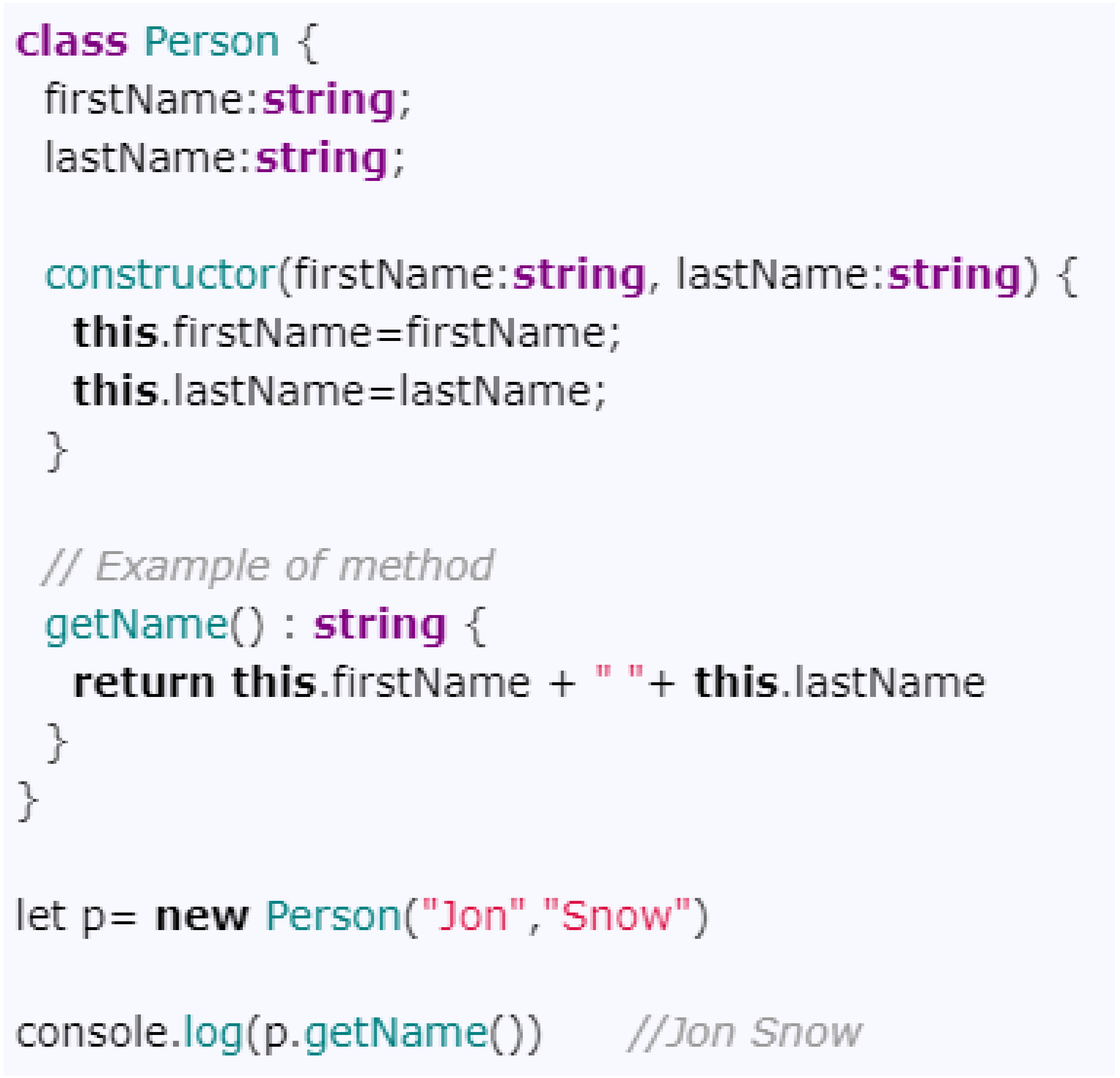


```
let p1= new Person("Jon","Snow")
let p2= new Person("Samwell","Tarly")

console.log(p1)  //Person: {  "firstName": "Jon",   "lastName": "Snow" }
console.log(p2)  //Person: {  "firstName": "Samwell",   "lastName": "Tarly" }
```

- The **constructor method creates the instance of the object** with the **properties defined in the class**.

Methods

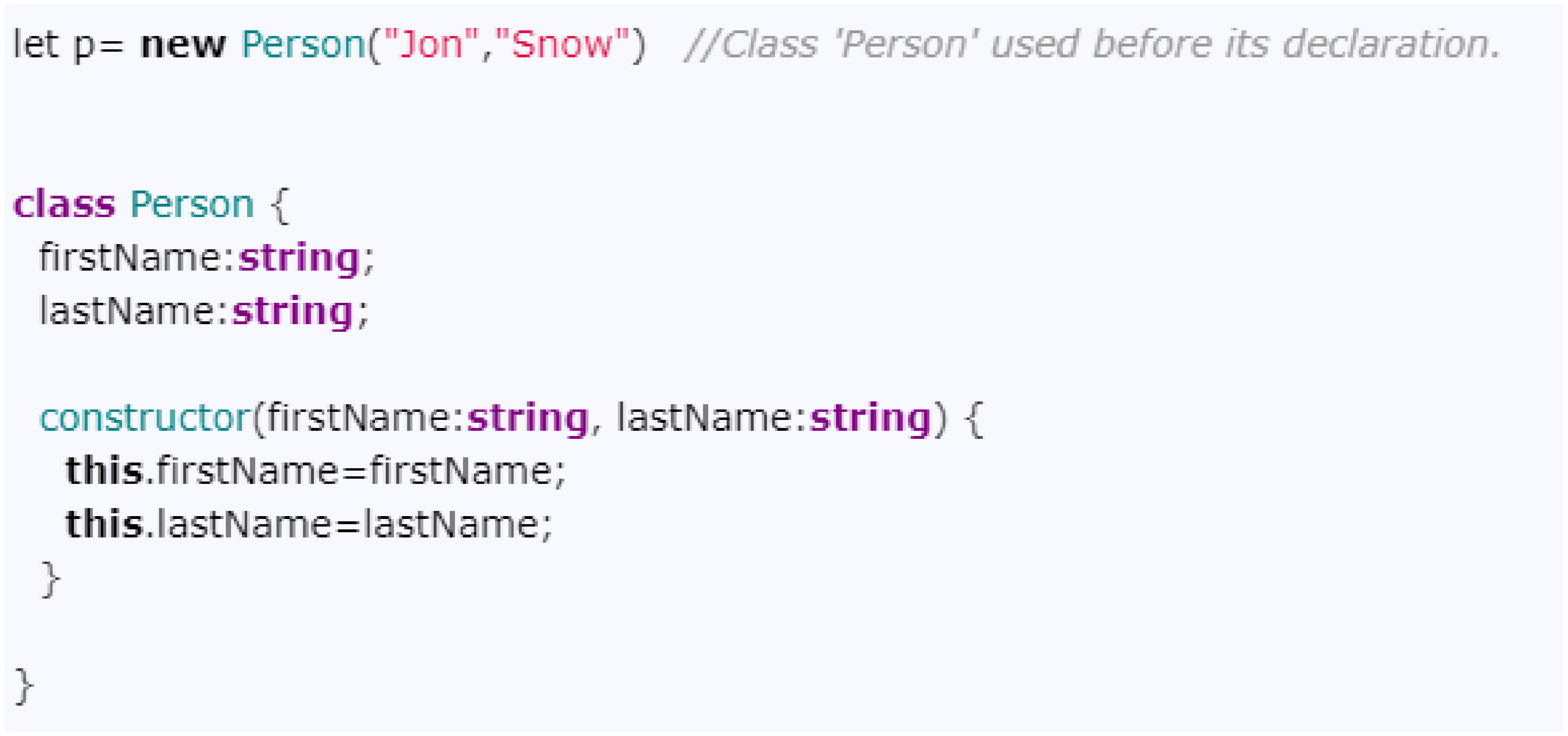
- A function declared in a class is called a ***method***. The **methods represent the behavior of the class**.
- Creating a method is similar to creating a function.



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

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

 // Example of method
 getName() : string {
  return this.firstName + " "+ this.lastName
 }
}

let p= new Person("Jon","Snow")

console.log(p.getName())     //Jon Snow
```

Class declarations are not hoisted

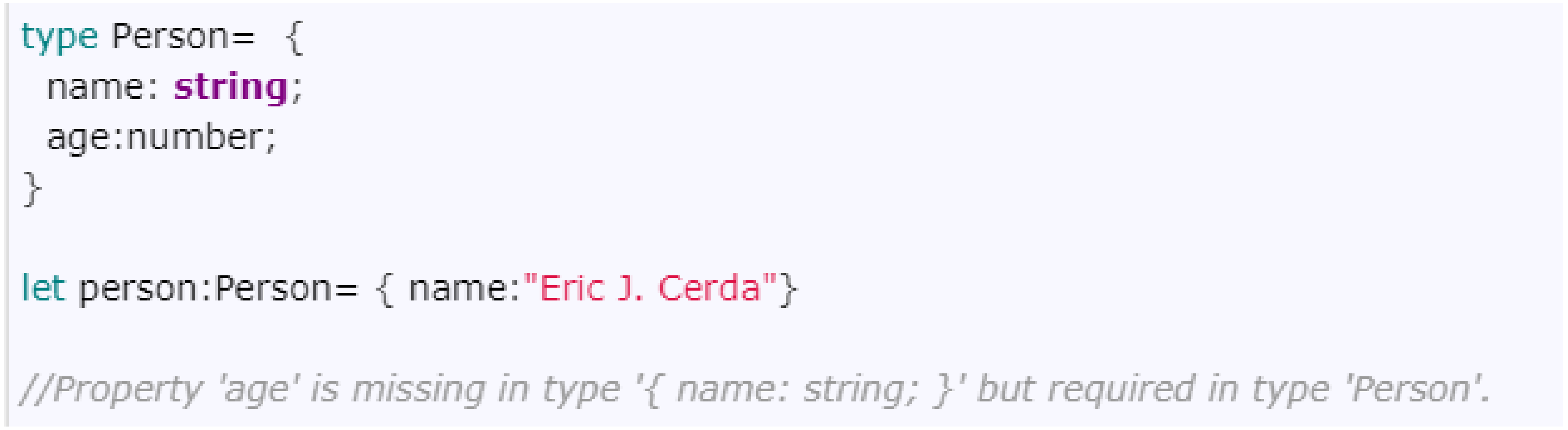
- The class declaration in TypeScript (and also in JavaScript) **is not hoisted.** i.e. **you cannot use class before you declare it.** The following code results in an error.



```
let p= new Person("Jon","Snow")   //Class 'Person' used before its declaration.



class Person {
  firstName:string;
  lastName:string;

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

}
```

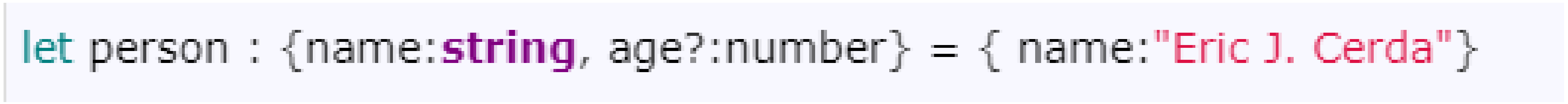
Optional Properties In TypeScript

- Optional Properties are those properties that are not mandatory.

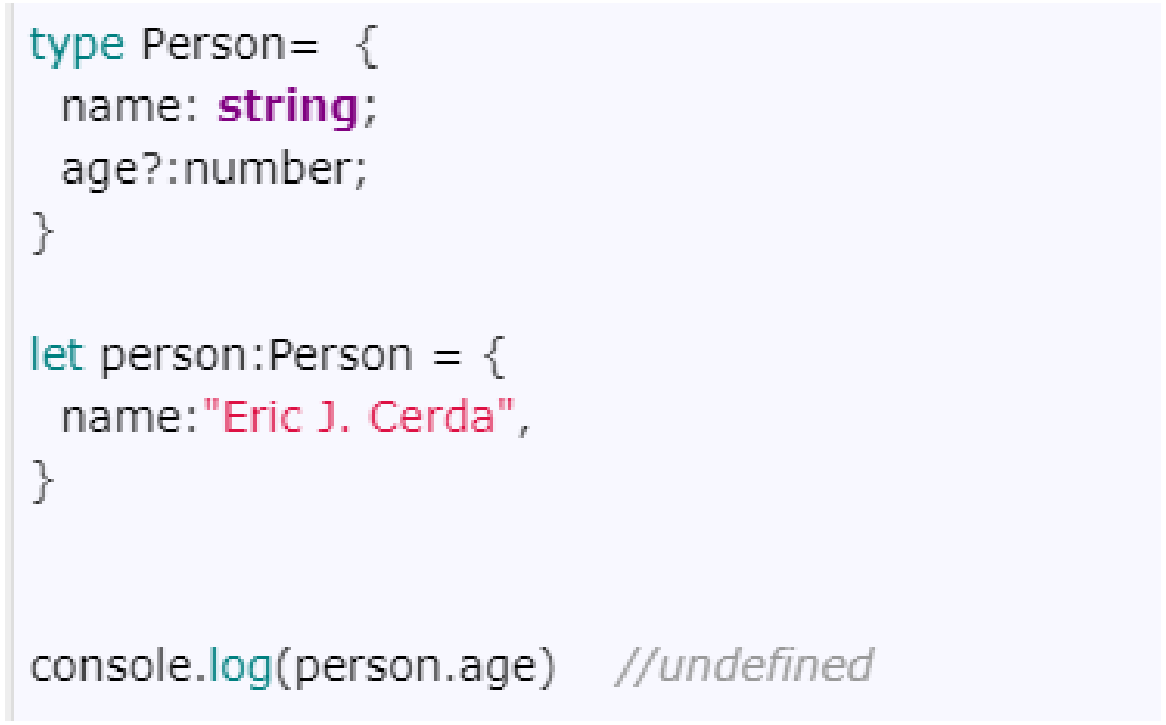


```
type Person=  {
  name: string;
  age:number;
}

let person:Person= { name:"Eric J. Cerda"}

//Property 'age' is missing in type '{ name: string; }' but required in type 'Person'.
```

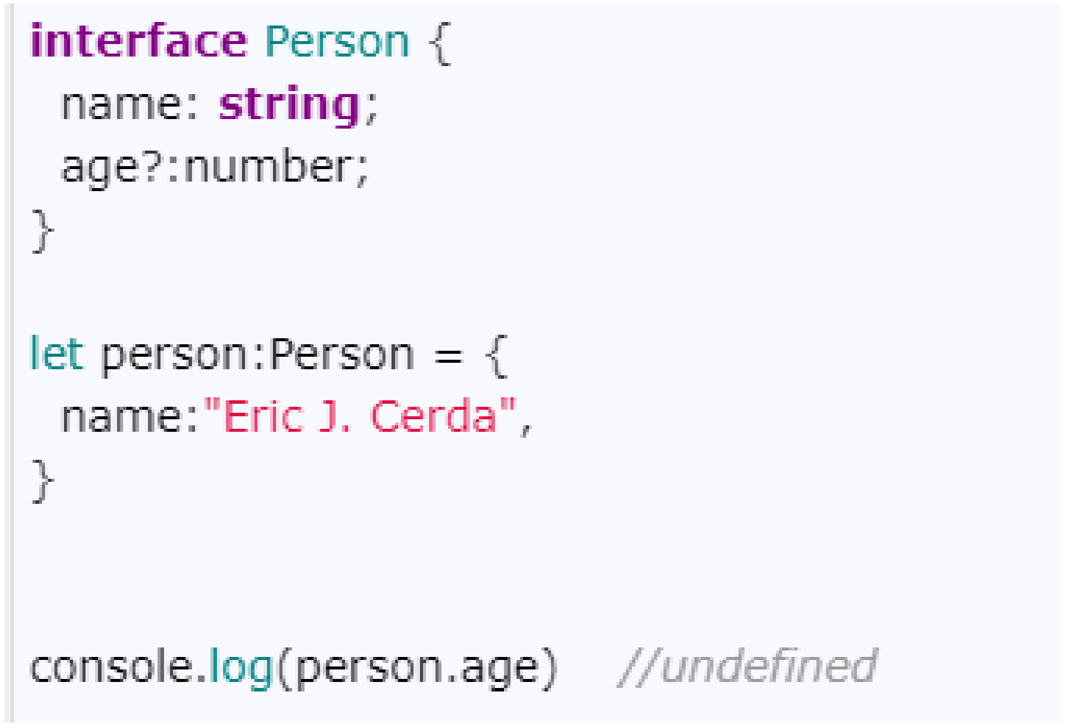
Creating Optional Properties in TypeScript



```
let person : {name:string, age?:number} = { name:"Eric J. Cerda"}
```

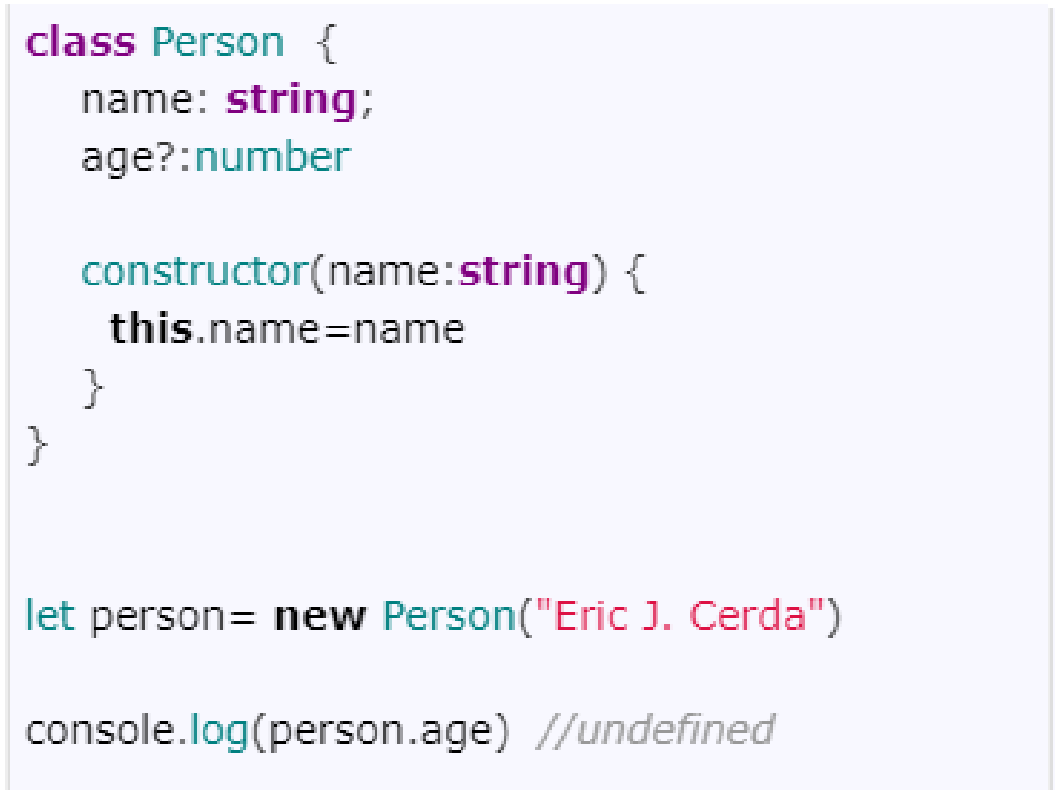


```
type Person=  {
  name: string;
  age?:number;
}

let person:Person = {
  name:"Eric J. Cerda",
}


console.log(person.age)   //undefined
```

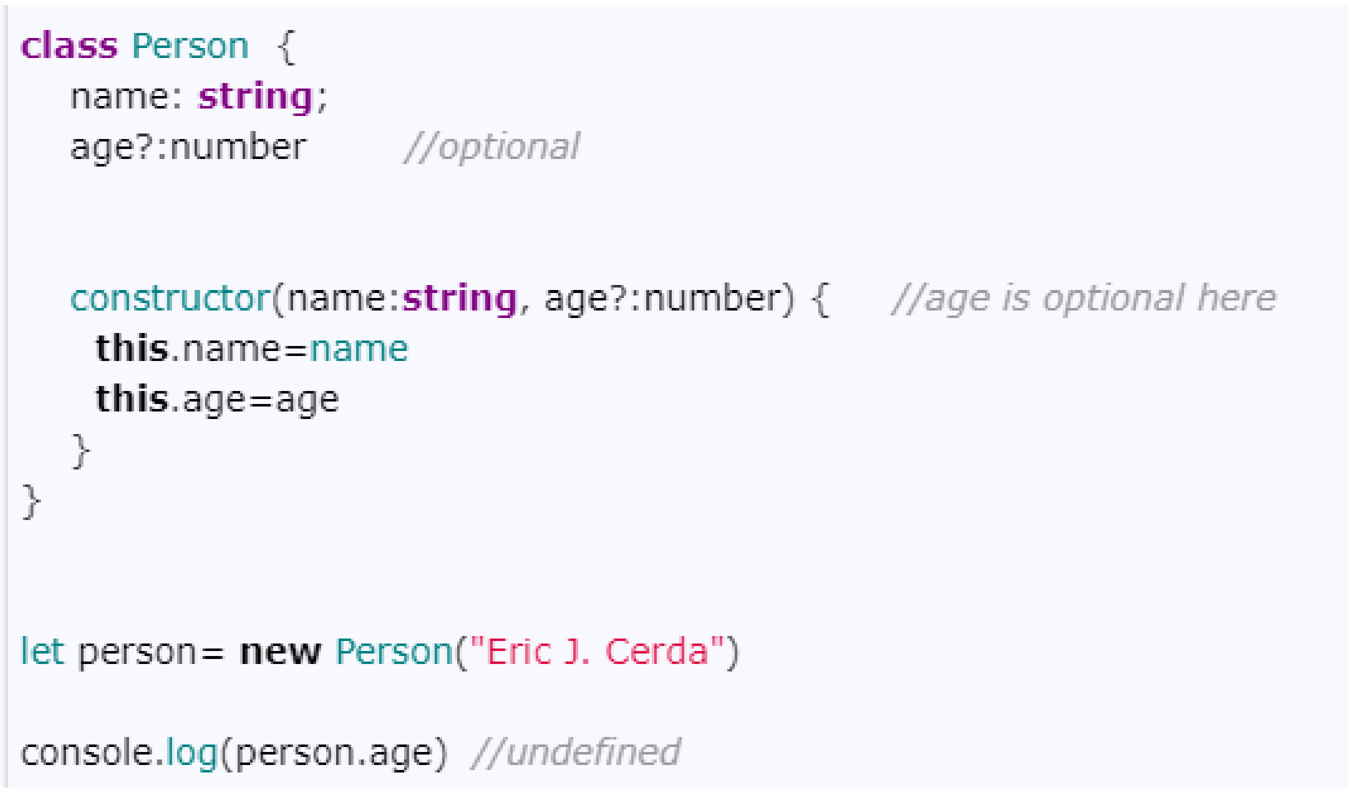


```
interface Person {
  name: string;
  age?:number;
}

let person:Person = {
  name:"Eric J. Cerda",
}


console.log(person.age)   //undefined
```

Creating Optional Properties in TypeScript



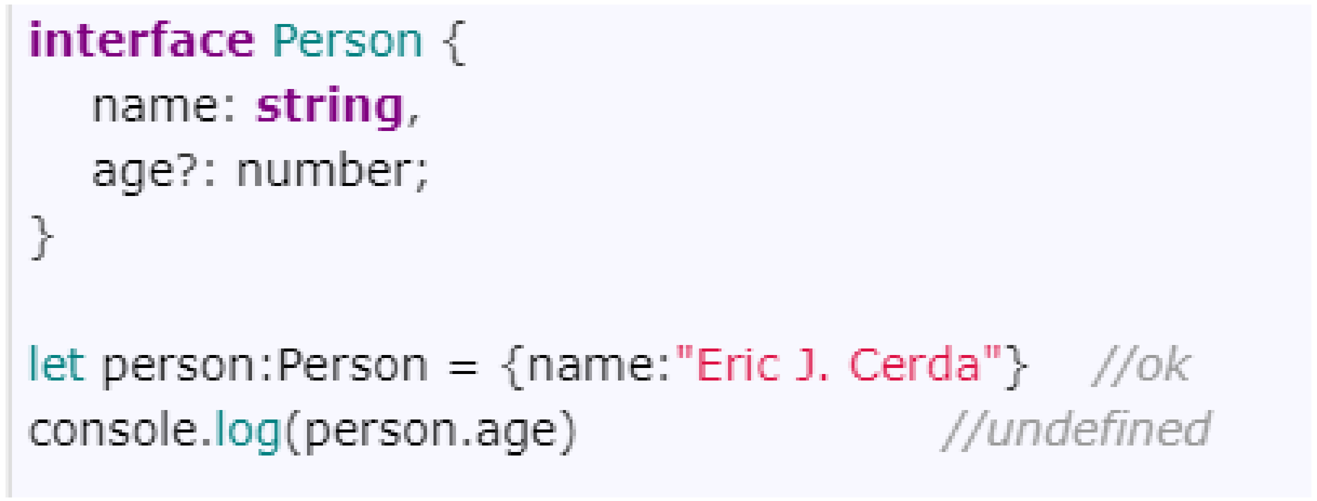
```
class Person  {
   name: string;
   age?:number

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


let person= new Person("Eric J. Cerda")

console.log(person.age)  //undefined
```



```
class Person  {
   name: string;
   age?:number       //optional


   constructor(name:string, age?:number) {    //age is optional here
     this.name=name
     this.age=age
   }
}


let person= new Person("Eric J. Cerda")

console.log(person.age)  //undefined
```

Optional Property and Undefined

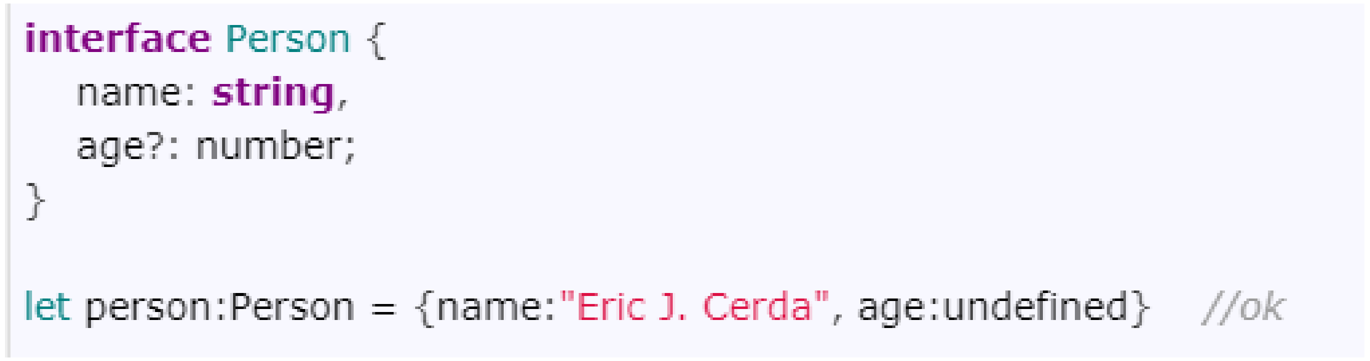
- Adding **?** makes **age** as optional property. Trying to access the value of **age** returns **undefined** as expected.



```
interface Person {
   name: string,
   age?: number;
}

let person:Person = {name:"Eric J. Cerda"}    //ok
console.log(person.age)                       //undefined
```

Optional Property and Undefined

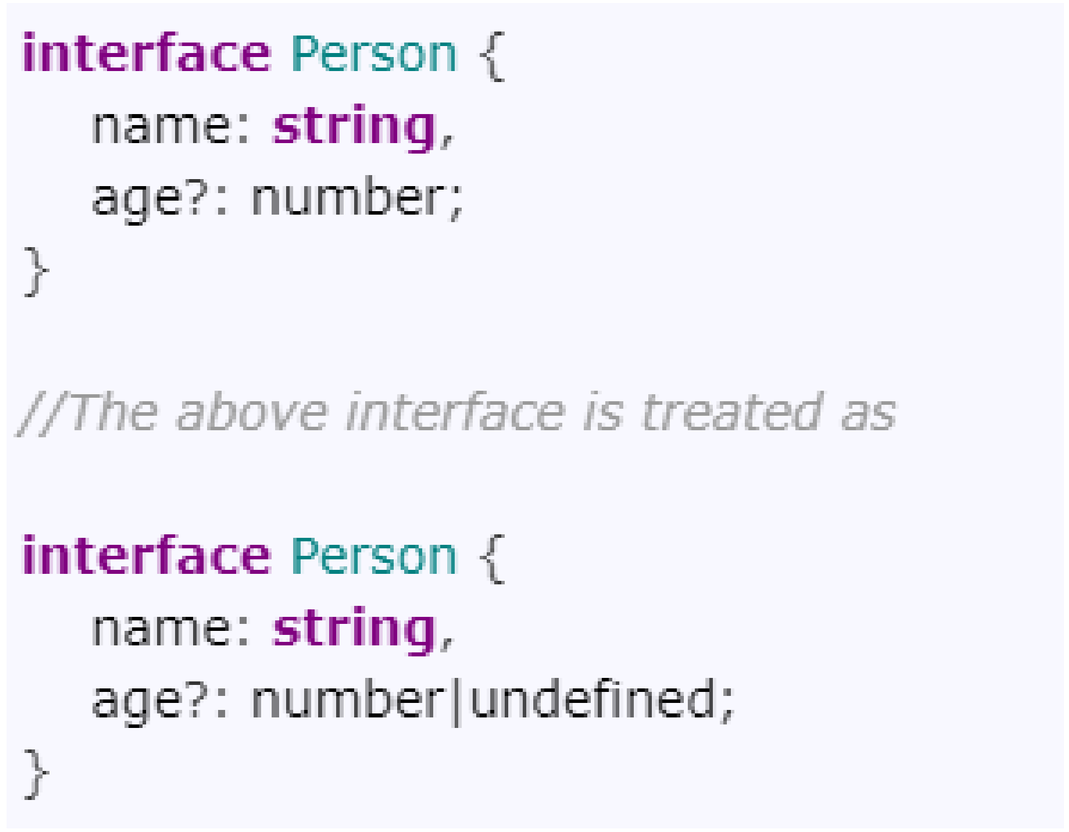
- But typescript also allows us to assign undefined value to an optional property although age is of type number.



```
interface Person {
   name: string,
   age?: number;
}

let person:Person = {name:"Eric J. Cerda", age:undefined}    //ok
```

Optional Property and Undefined

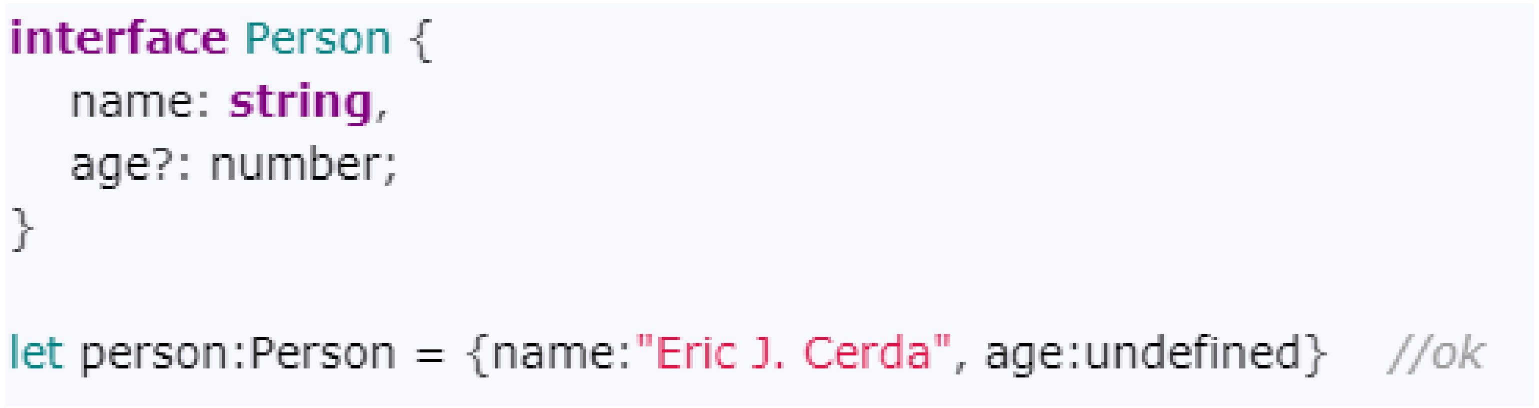
- This is because TypeScript treats every optional property as a union type of undefined and assigned type.



```
interface Person {
   name: string,
   age?: number;
}

//The above interface is treated as

interface Person {
   name: string,
   age?: number|undefined;
}
```

ExactOptionalPropertyTypes Config Option

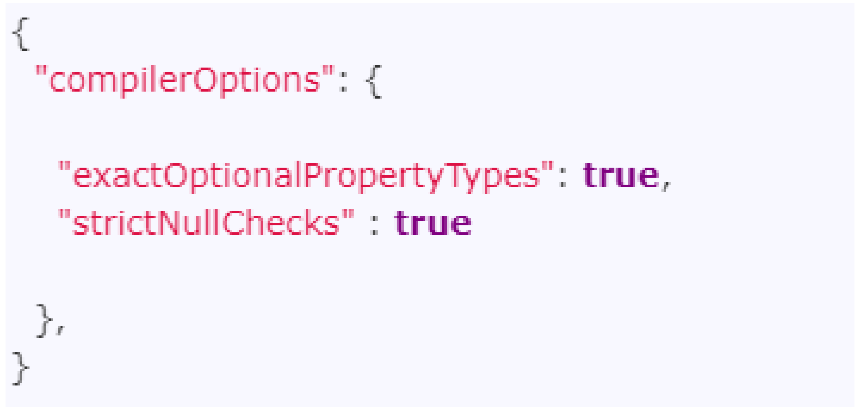
- In the previous section, we discussed that typescript allows us to assign undefined to an optional property.



```
interface Person {
   name: string,
   age?: number;
}

let person:Person = {name:"Eric J. Cerda", age:undefined}   //ok
```

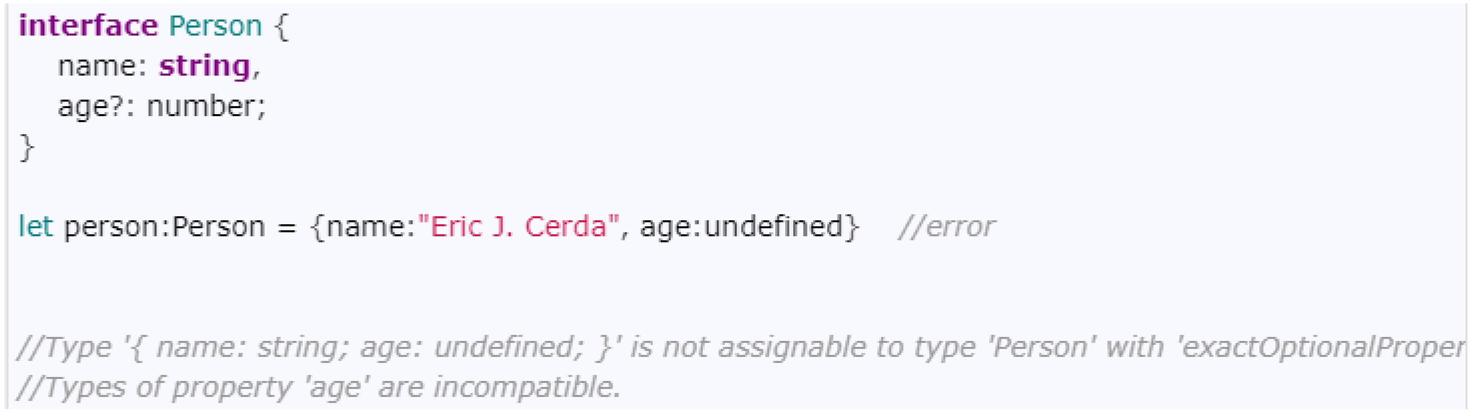
- You can stop that by setting the **exactOptionalPropertyTypes** to true in the **tsconfig.json** file under the section **compilerOptions**. Note that you also need to enable **strictNullChecks** for the **exactOptionalPropertyTypes** to work.

ExactOptionalPropertyTypes Config Option



```
{
 "compilerOptions": {

   "exactOptionalPropertyTypes": true,
   "strictNullChecks" : true

 },
}
```

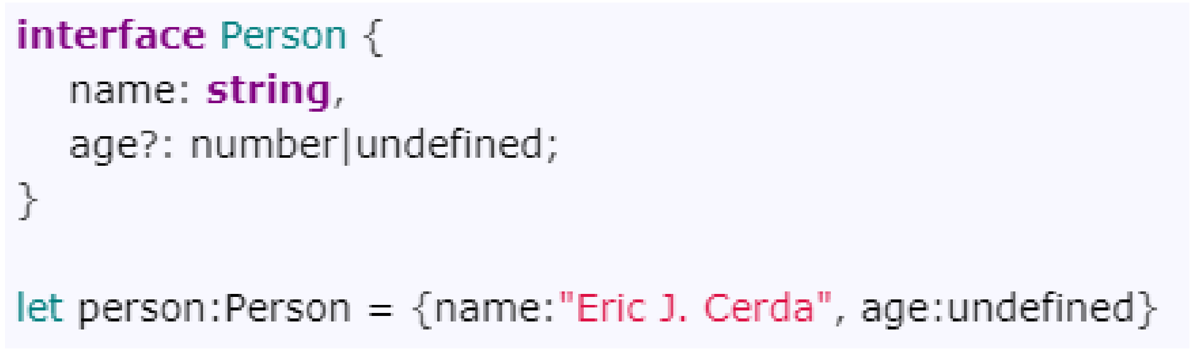
- With **exactOptionalPropertyTypes** enabled, the TypeScript compiler throws an error when we try to assign the undefined to the age variable.



```
interface Person {
   name: string,
   age?: number;
}

let person:Person = {name:"Eric J. Cerda", age:undefined}    //error



//Type '{ name: string; age: undefined; }' is not assignable to type 'Person' with 'exactOptionalProper
//Types of property 'age' are incompatible.
```

ExactOptionalPropertyTypes Config Option

- You can override the behavior by explicitly setting the type as `undefined`



```
interface Person {
   name: string,
   age?: number|undefined;
}

let person:Person = {name:"Eric J. Cerda", age:undefined}
```

Make all properties optional of an existing type

- Typescript utility types allow us to construct a new type from an existing type. The **Partial utility** type is one such utility that takes a type and constructs a new type with all the properties set to optional.

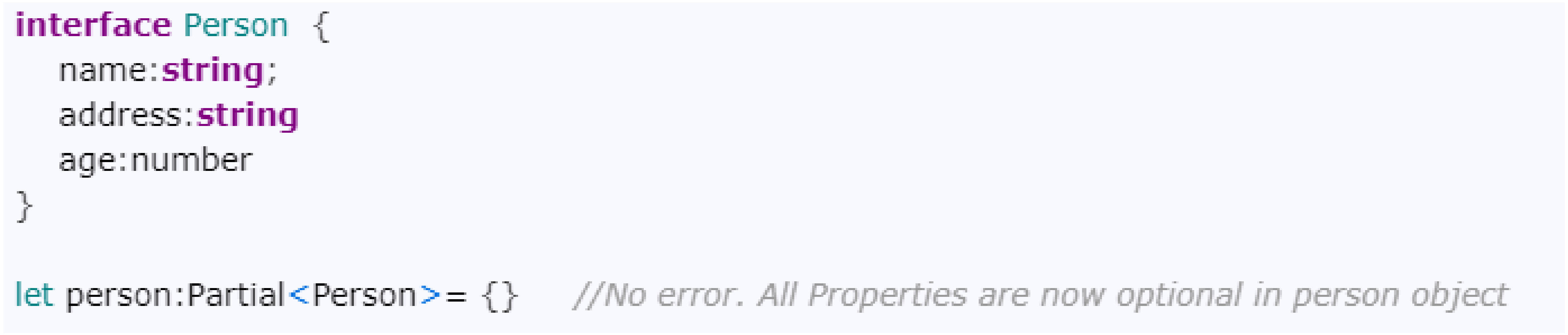
```
Partial<exitingType>
```

- In this example, the Person interface has all properties marked as required. Assigning it an empty object will result in an error.

```
interface Person  {
   name:string;
   address:string
   age:number
}

let person:Person= {}
//Type '{}' is missing the following properties from type 'Person': name, age
```

Make all properties optional of an existing type

- You can use the `Partial<Person>` to create a new type with all properties set to optional. Hence it will not result in any compiler error.



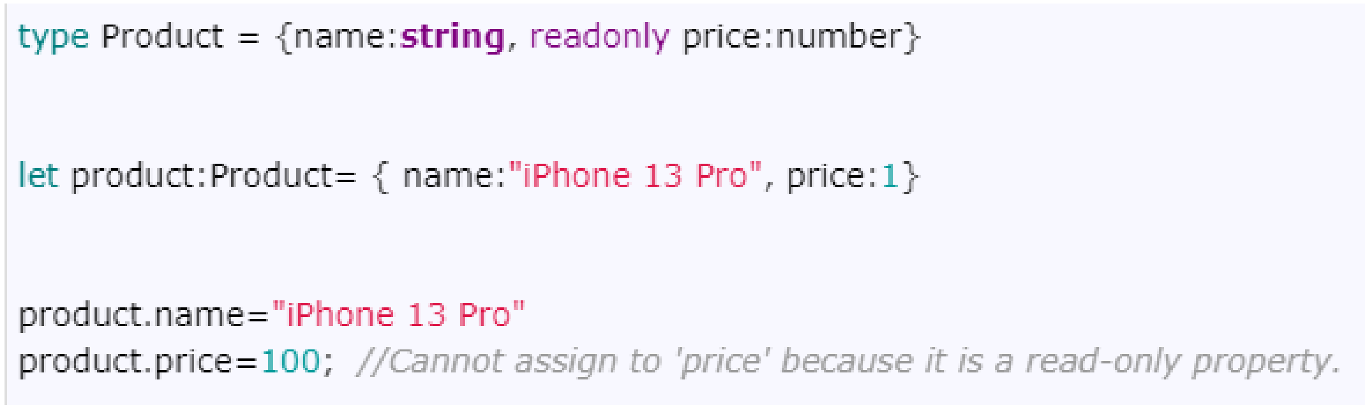
```
interface Person  {
   name:string;
   address:string
   age:number
}

let person:Partial<Person>= {}     //No error. All Properties are now optional in person object
```

Creating Read-Only Properties in TypeScript

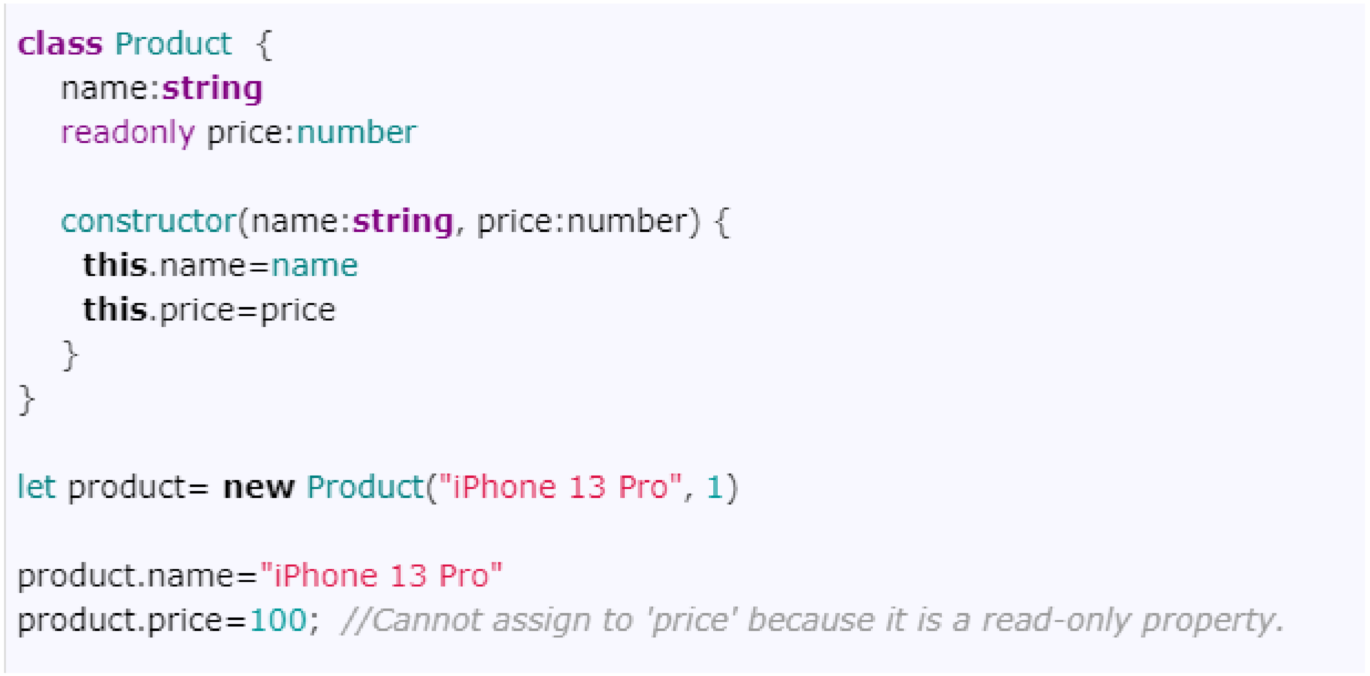
```
let product : {name:string, readonly price:number} = { name:"iPhone 13 Pro", price:1}

product.name="iPhone 13 Pro"  //ok
product.price=100;  //Cannot assign to 'price' because it is a read-only property.
```

```
interface Product  {name:string, readonly price:number}


let product:Product= { name:"iPhone 13 Pro", price:1}


product.name="iPhone 13 Pro"
product.price=100;  //Cannot assign to 'price' because it is a read-only property.
```

Creating Read-Only Properties in TypeScript



```
type Product = {name:string, readonly price:number}


let product:Product= { name:"iPhone 13 Pro", price:1}


product.name="iPhone 13 Pro"
product.price=100;  //Cannot assign to 'price' because it is a read-only property.
```



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

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

let product= new Product("iPhone 13 Pro", 1)

product.name="iPhone 13 Pro"
product.price=100;  //Cannot assign to 'price' because it is a read-only property.
```

Creating Read-Only Properties in TypeScript

- Readonly exists only in compile time

  The **readonly** is specific to TypeScript and does not exist in run time. Typescript compiler uses it to check for illegal property assignments in compile time. So, the variable does not have any protection in the run-time. Once the code is transpiled into JavaScript **readonly** is gone.

- Readonly vs. const
  - const apply to variables only. readonly for properties
  - const is runtime check readonly is compile-time check

Private, Public & Protected Access Modifiers in TypeScript

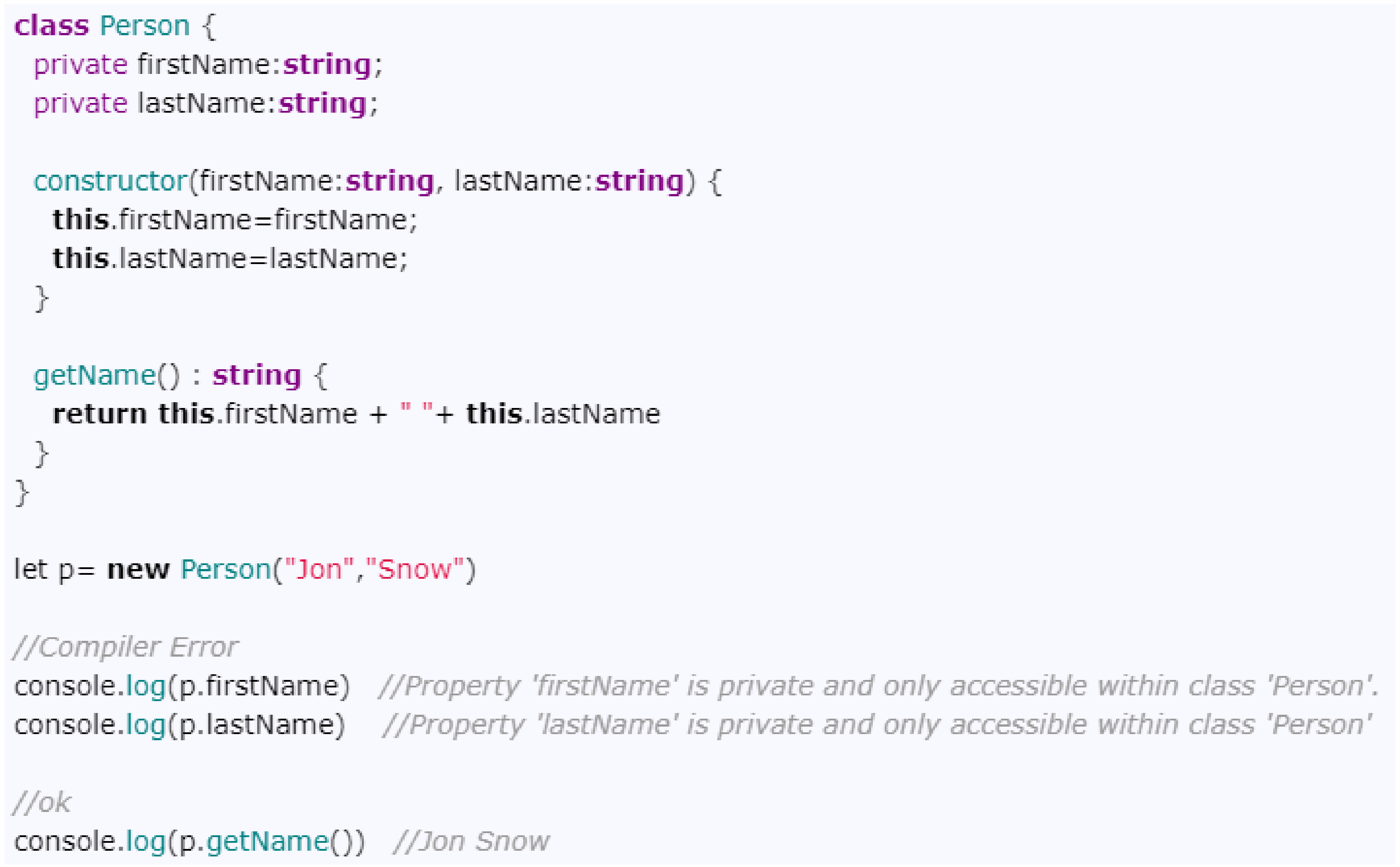
- Private, Public & Protected are the Access Modifiers that we can use in Typescript to control whether certain methods or properties are visible to code outside the class.
- **Public access modifier** allows the class properties and method freely accessible anywhere in the code. The public is the default access modifier if we do not specify the access modifier

Public

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

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

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

let p= new Person("Jon","Snow")


console.log(p.firstName)   //Jon
console.log(p.lastName)    //Snow
console.log(p.getName())     //Jon Snow
```

Private

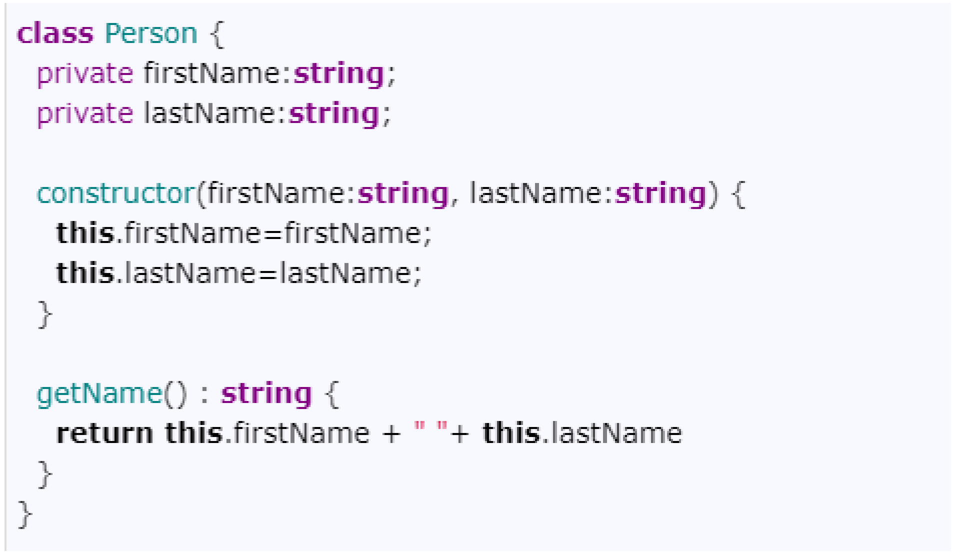
- The Private access modifier restricts access to the class member from within the class only. We cannot access the property or the method from outside the class



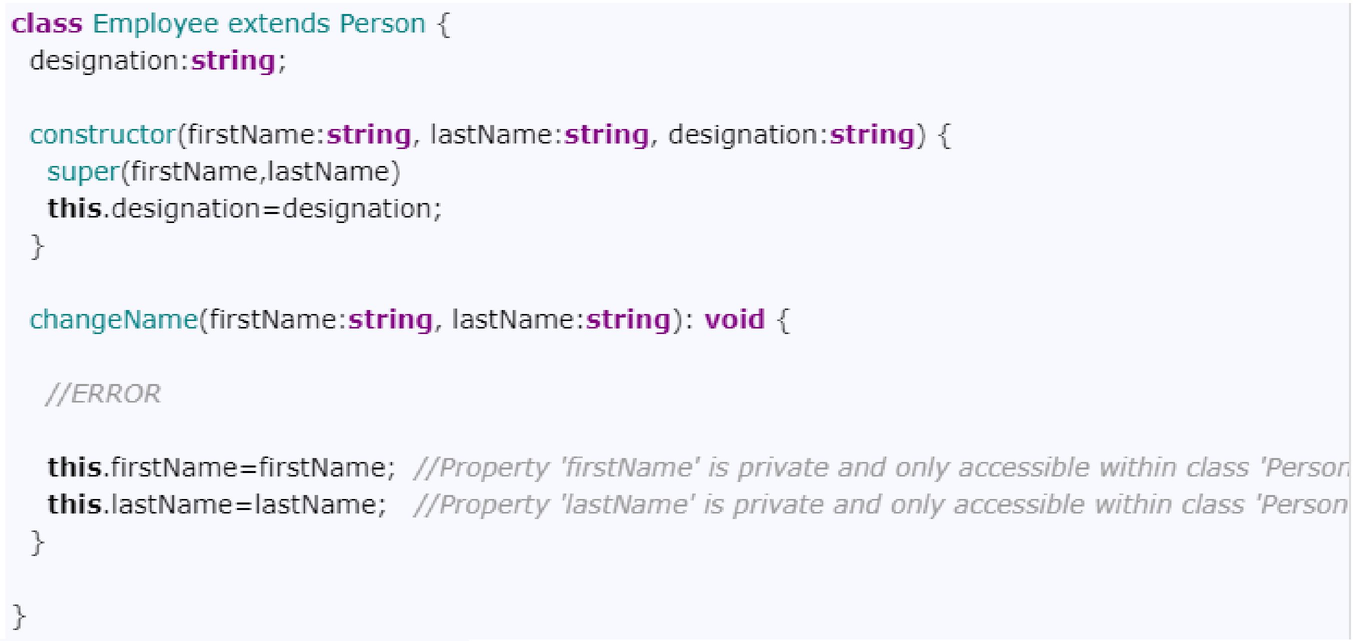
```
class Person {
  private firstName:string;
  private lastName:string;

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

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

let p= new Person("Jon","Snow")

//Compiler Error
console.log(p.firstName)   //Property 'firstName' is private and only accessible within class 'Person'.
console.log(p.lastName)    //Property 'lastName' is private and only accessible within class 'Person'

//ok
console.log(p.getName())   //Jon Snow
```

Private

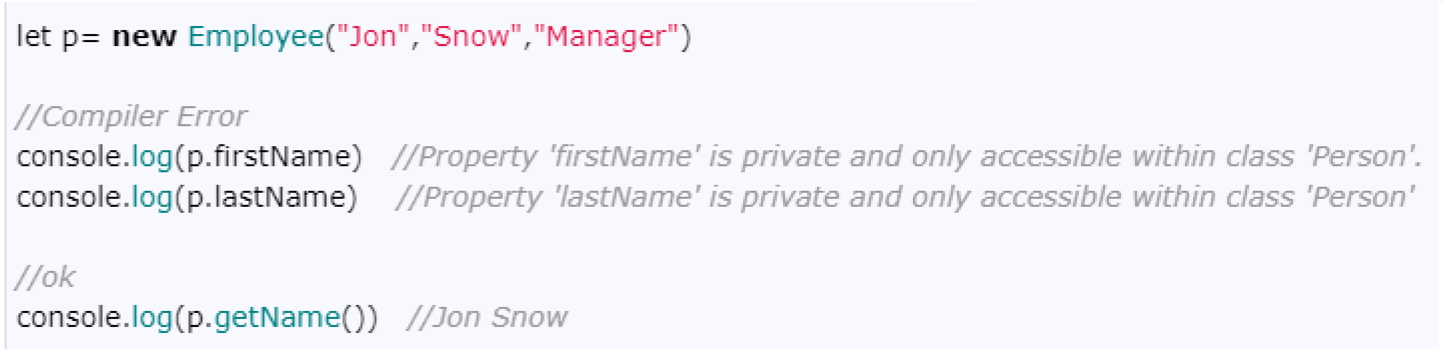
- We cannot even access the private property in a derived class.



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

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

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



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

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

  changeName(firstName:string, lastName:string): void {

    //ERROR

    this.firstName=firstName;  //Property 'firstName' is private and only accessible within class 'Person
    this.lastName=lastName;    //Property 'lastName' is private and only accessible within class 'Person
  }

}
```



```
let p= new Employee("Jon","Snow","Manager")

//Compiler Error
console.log(p.firstName)   //Property 'firstName' is private and only accessible within class 'Person'.
console.log(p.lastName)    //Property 'lastName' is private and only accessible within class 'Person'

//ok
console.log(p.getName())   //Jon Snow
```

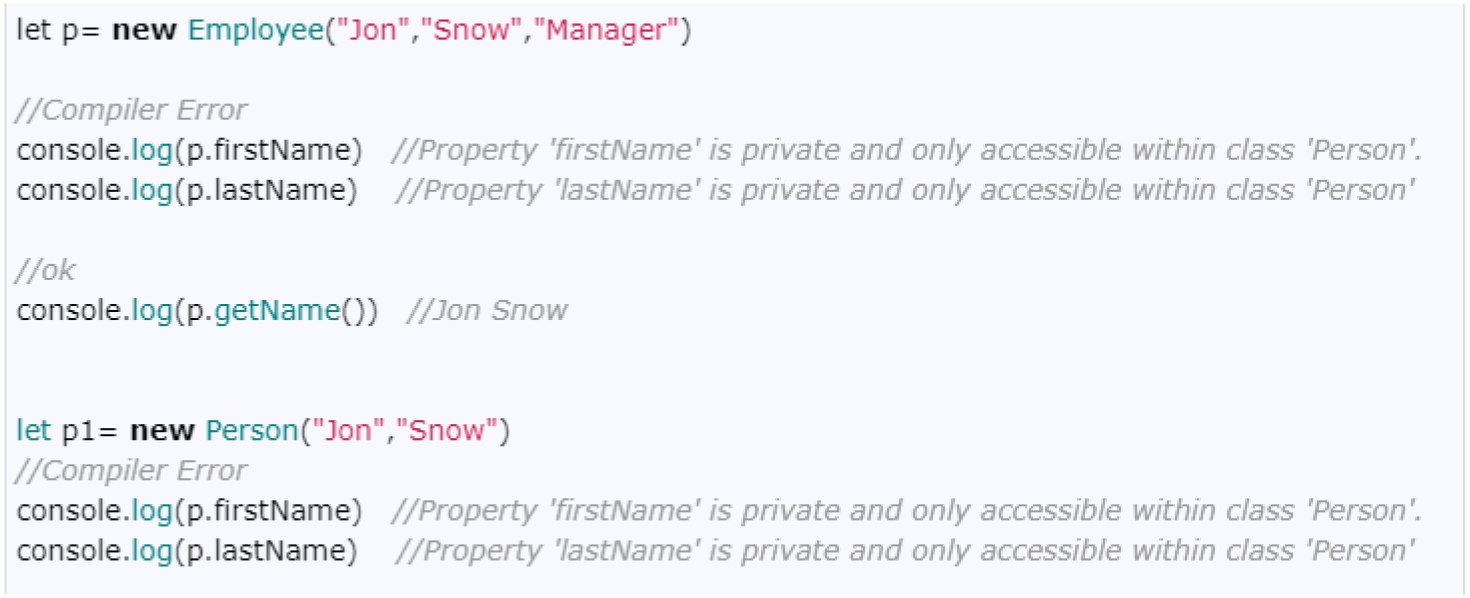
Protected

- The Protected modifier allows access to the class member from itself and from any classes that inherit (sub-class) from it.

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

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

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

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

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

  changeName(firstName:string, lastName:string): void {

    //Ok
    this.firstName=firstName;
    this.lastName=lastName;
  }

}
```

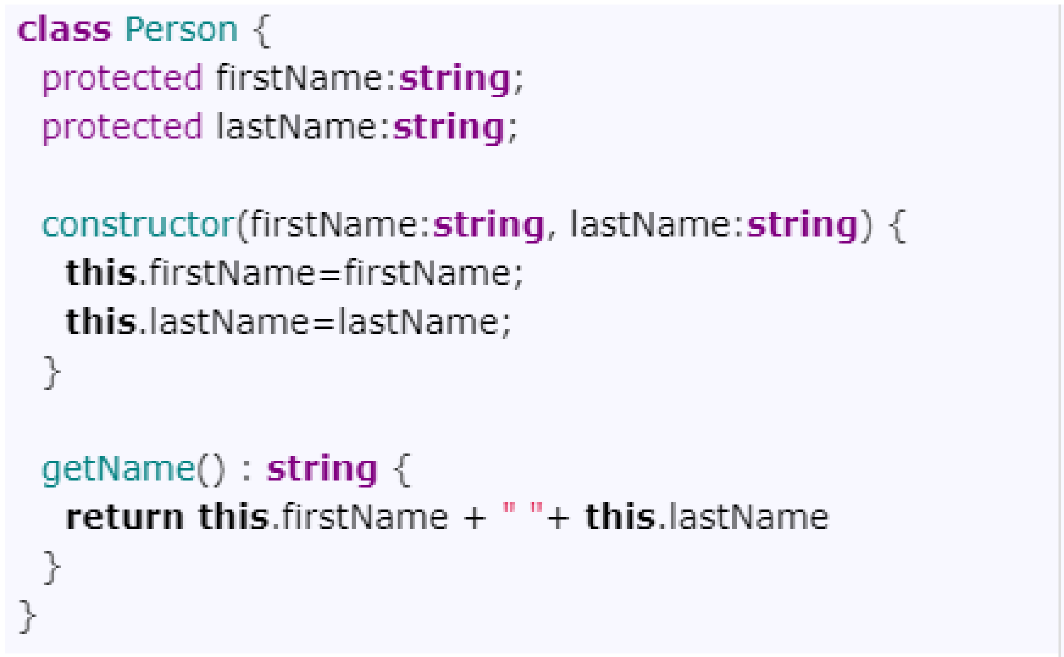
Protected



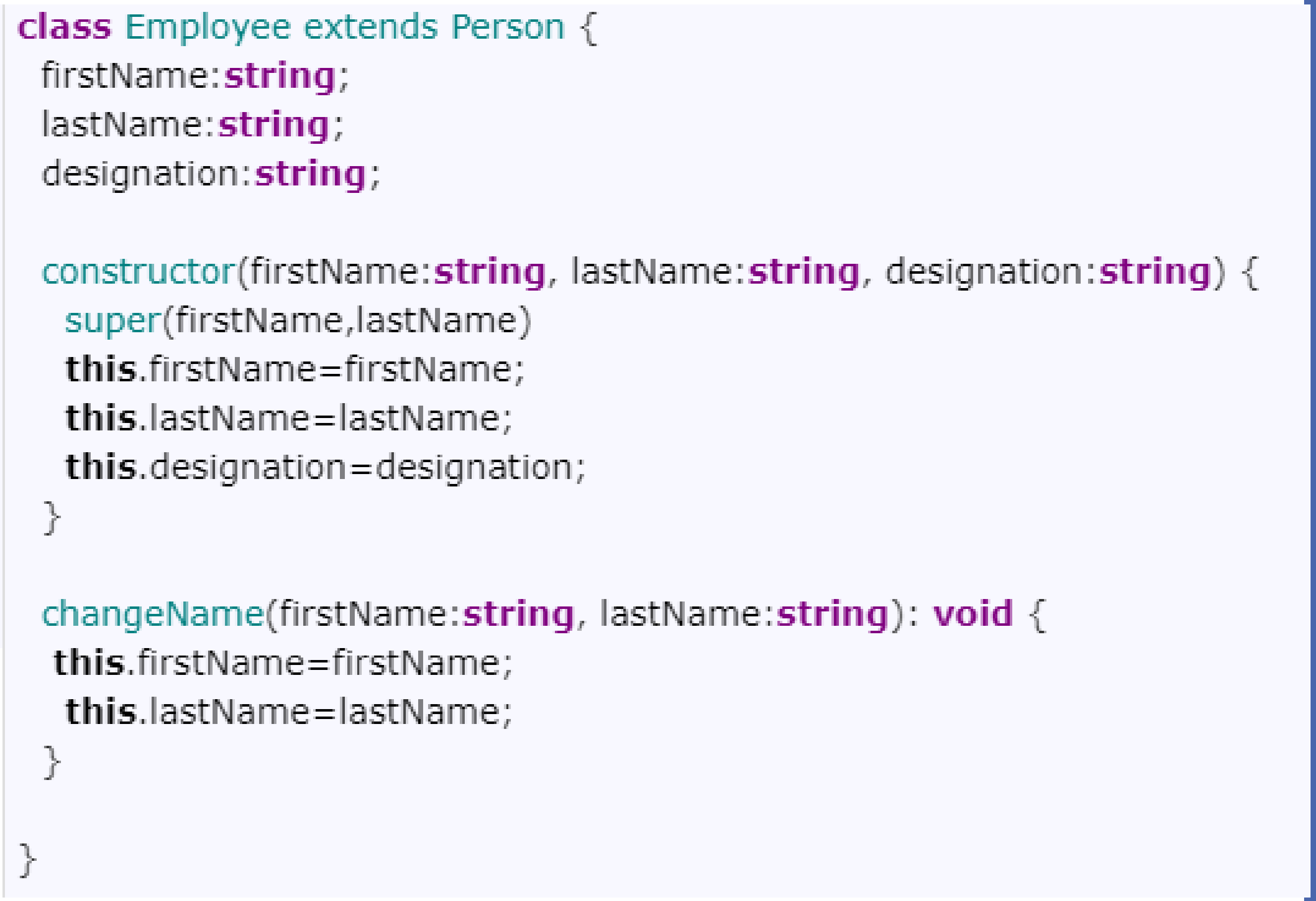
```
let p= new Employee("Jon","Snow","Manager")

//Compiler Error
console.log(p.firstName)   //Property 'firstName' is private and only accessible within class 'Person'.
console.log(p.lastName)    //Property 'lastName' is private and only accessible within class 'Person'

//ok
console.log(p.getName())   //Jon Snow


let p1= new Person("Jon","Snow")
//Compiler Error
console.log(p.firstName)   //Property 'firstName' is private and only accessible within class 'Person'.
console.log(p.lastName)    //Property 'lastName' is private and only accessible within class 'Person'
```

Overriding Access Modifier in the Derived Class

- The derived class can redeclare the property and override the Access modifier.

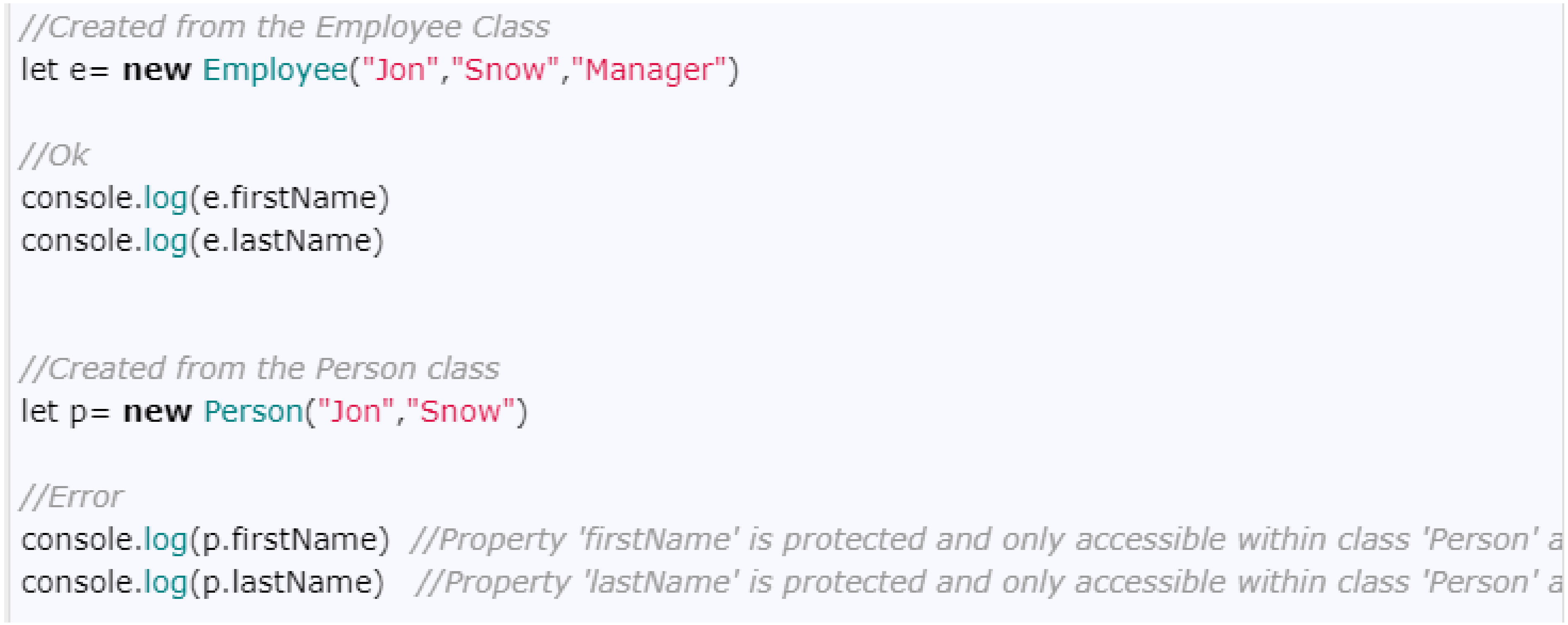


```
class Person {
  protected firstName:string;
  protected lastName:string;

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

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



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

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

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

}
```

Overriding Access Modifier in the Derived Class



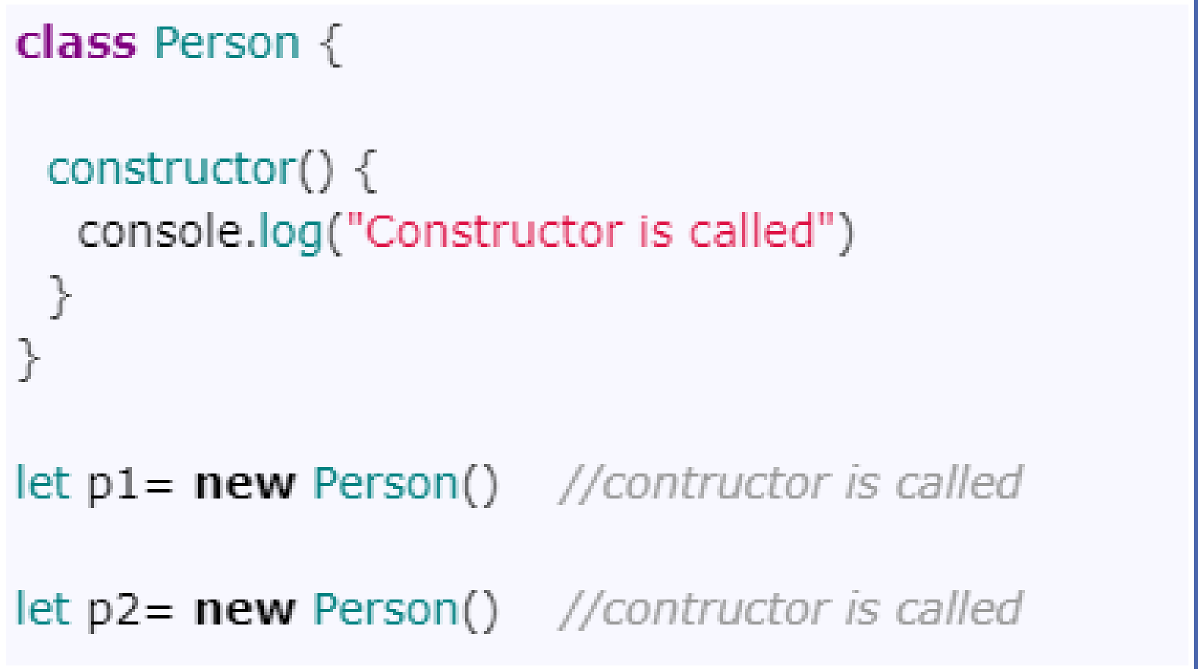
```
//Created from the Employee Class
let e= new Employee("Jon","Snow","Manager")

//Ok
console.log(e.firstName)
console.log(e.lastName)



//Created from the Person class
let p= new Person("Jon","Snow")

//Error
console.log(p.firstName)  //Property 'firstName' is protected and only accessible within class 'Person' a
console.log(p.lastName)   //Property 'lastName' is protected and only accessible within class 'Person' a
```

Constructor in TypeScript

- A constructor is **a special function of the class that is** automatically invoked when we create an instance of the class in Typescript.
- We use it to **initialize the properties** of the current instance of the class.
- Using the **constructor parameter properties** or **Parameter shorthand syntax**, we can **add new properties** to the class.
- We can also create **multiple constructors** using the technique of **constructor method overload**.

Creating a Class Constructor

- The constructor method in a class must have the name **constructor**.
- A class can have **only one implementation of the constructor method**.
- The constructor method **is invoked every time** we create an instance from the class using the new operator.
- It always returns the newly created object.

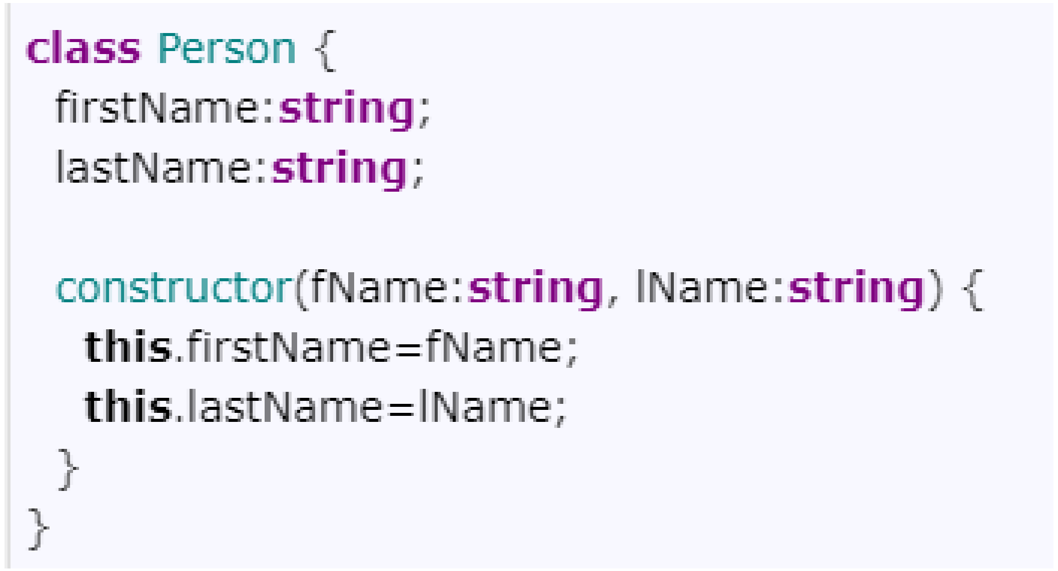


```
class Person {

  constructor() {
    console.log("Constructor is called")
  }
}

let p1= new Person()    //contructor is called

let p2= new Person()    //contructor is called
```

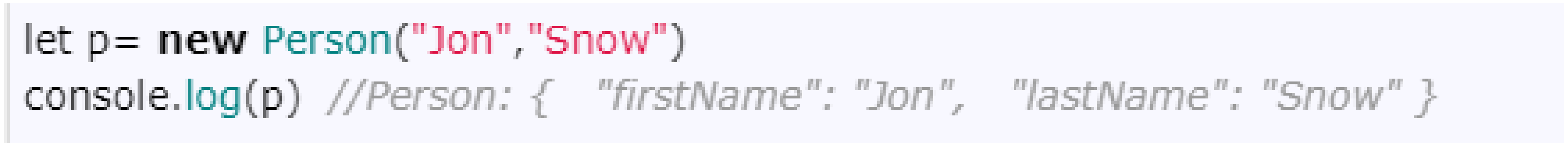
The constructor method creates the instance of the object with the property defined in the class. We can access the current instance using the **this** inside the constructor. Once finished constructor function returns the new object.

Parameters to the Constructor method

- Constructors are just like normal functions in the way that they also accept parameters. We need to pass parameter values when we create a new class instance.

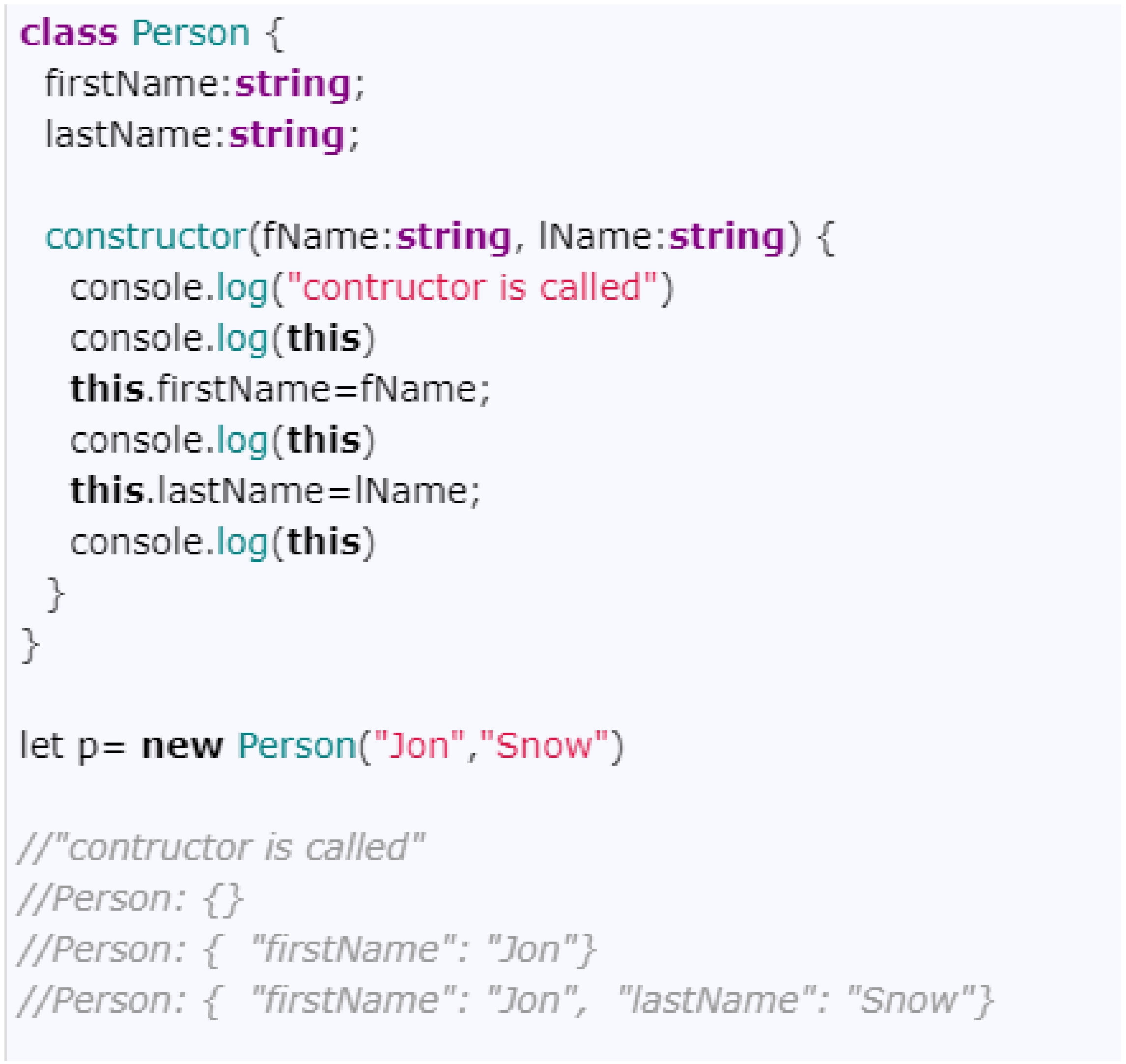


```
class Person {
  firstName:string;
  lastName:string;

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



```
let p= new Person("Jon","Snow")
console.log(p) //Person: {  "firstName": "Jon",   "lastName": "Snow" }
```

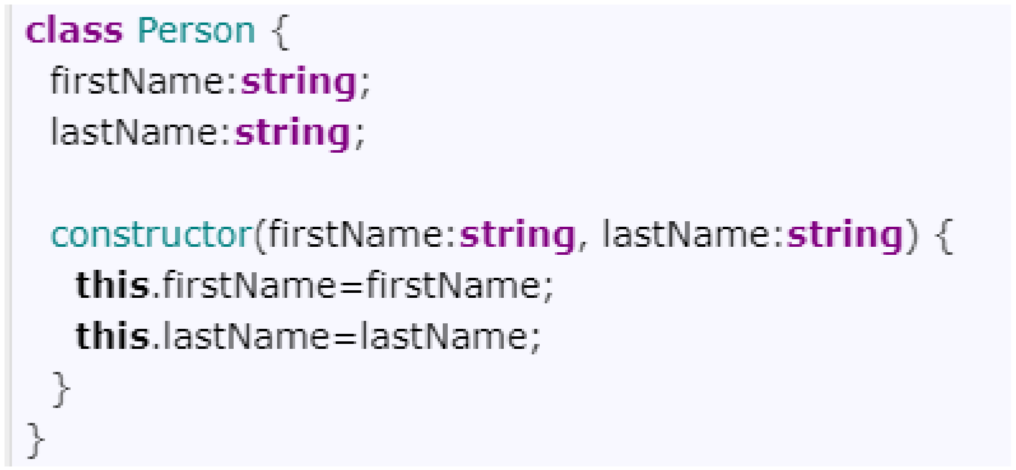
Parameters to the Constructor method



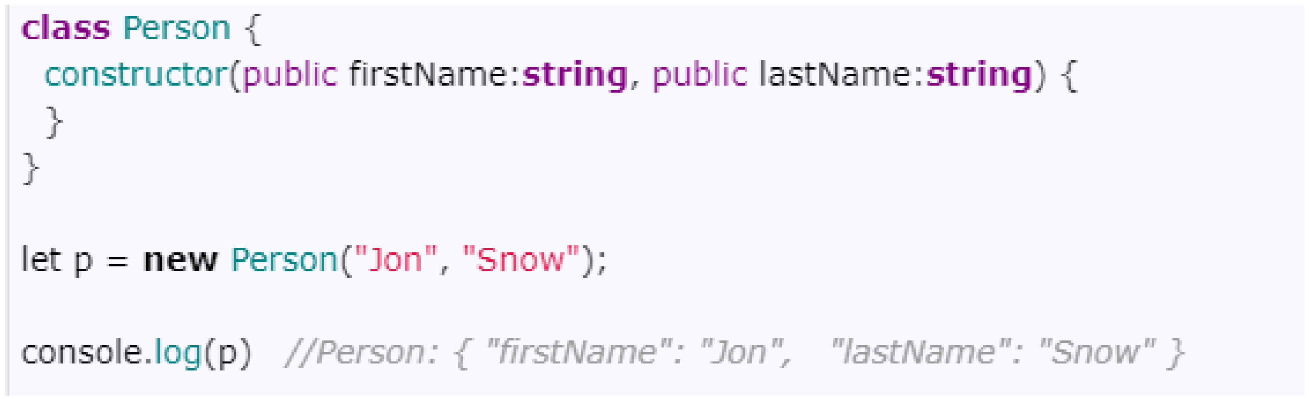
```
class Person {
  firstName:string;
  lastName:string;

  constructor(fName:string, lName:string) {
    console.log("contructor is called")
    console.log(this)
    this.firstName=fName;
    console.log(this)
    this.lastName=lName;
    console.log(this)
  }
}

let p= new Person("Jon","Snow")

//"contructor is called"
//Person: {}
//Person: {  "firstName": "Jon"}
//Person: {  "firstName": "Jon",  "lastName": "Snow"}
```

Constructor Parameter Properties

- Constructor Parameter Properties (or Property Shorthand syntax) offers a special shorthand syntax to **convert parameters of constructor function into properties**. We can do this by prefixing a constructor parameter with one of the visibility modifiers



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

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



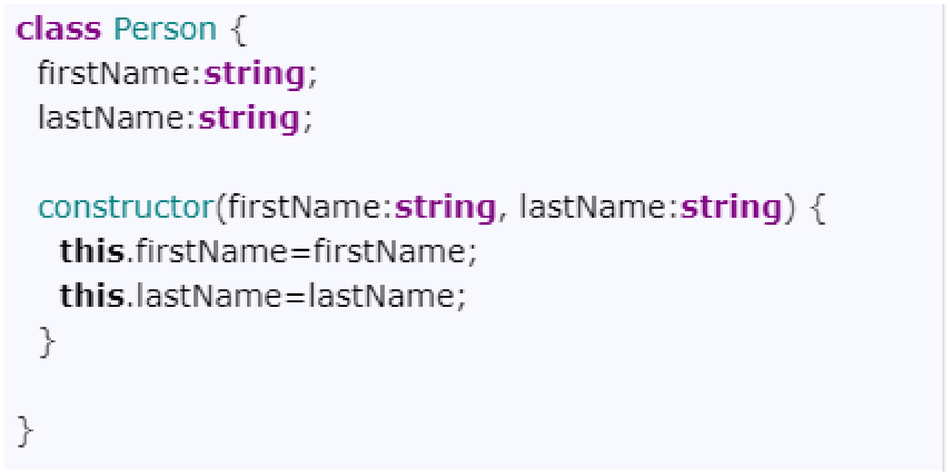
```
class Person {
  constructor(public firstName:string, public lastName:string) {
  }
}

let p = new Person("Jon", "Snow");

console.log(p)  //Person: { "firstName": "Jon",   "lastName": "Snow" }
```

Passing Default Values in the Constructor method

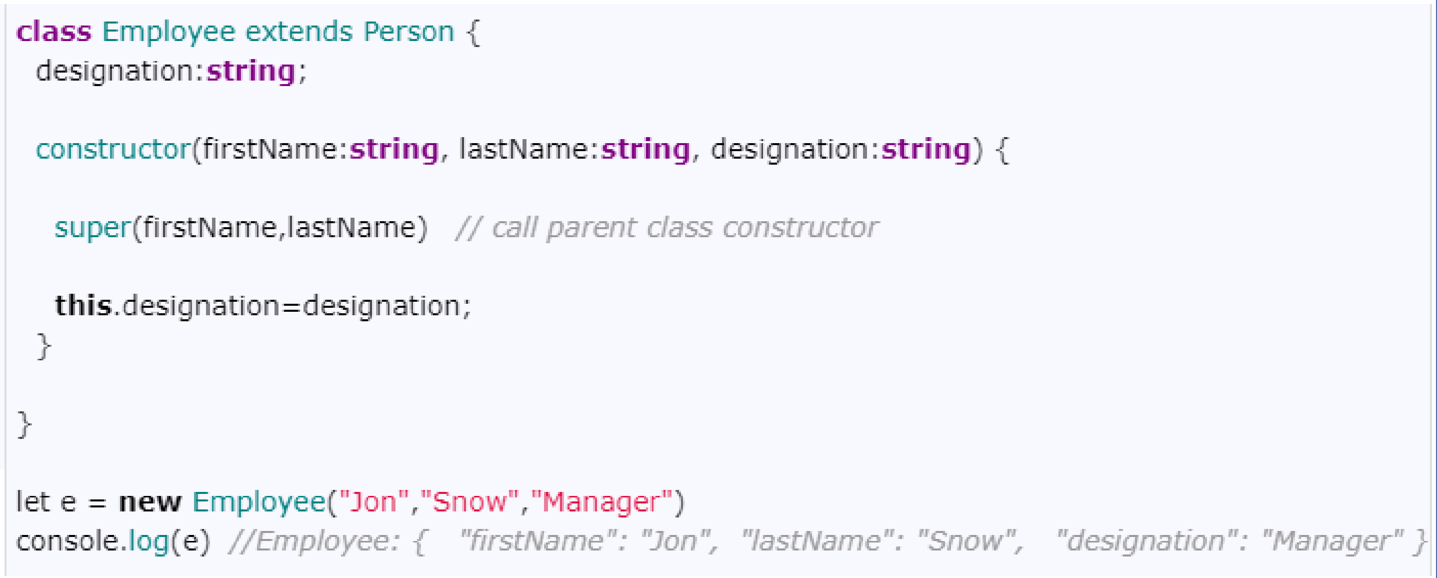
```
class Point {
  x: number;
  y: number;

  constructor(x = 10, y = 20) {
    this.x = x;
    this.y = y;
  }
}

//Using default Values
let p1 = new Point();
console.log(p1) //Point: {   "x": 10,   "y": 20 }



let p2 = new Point(100,200);
console.log(p2) //Point: {   "x": 100,   "y": 200 }
```

Constructor functions in the derived class

- The derived class must call the constructor of the parent class (base class). We do this by invoking the **super** method. We must call the super method before we use **this** variable.



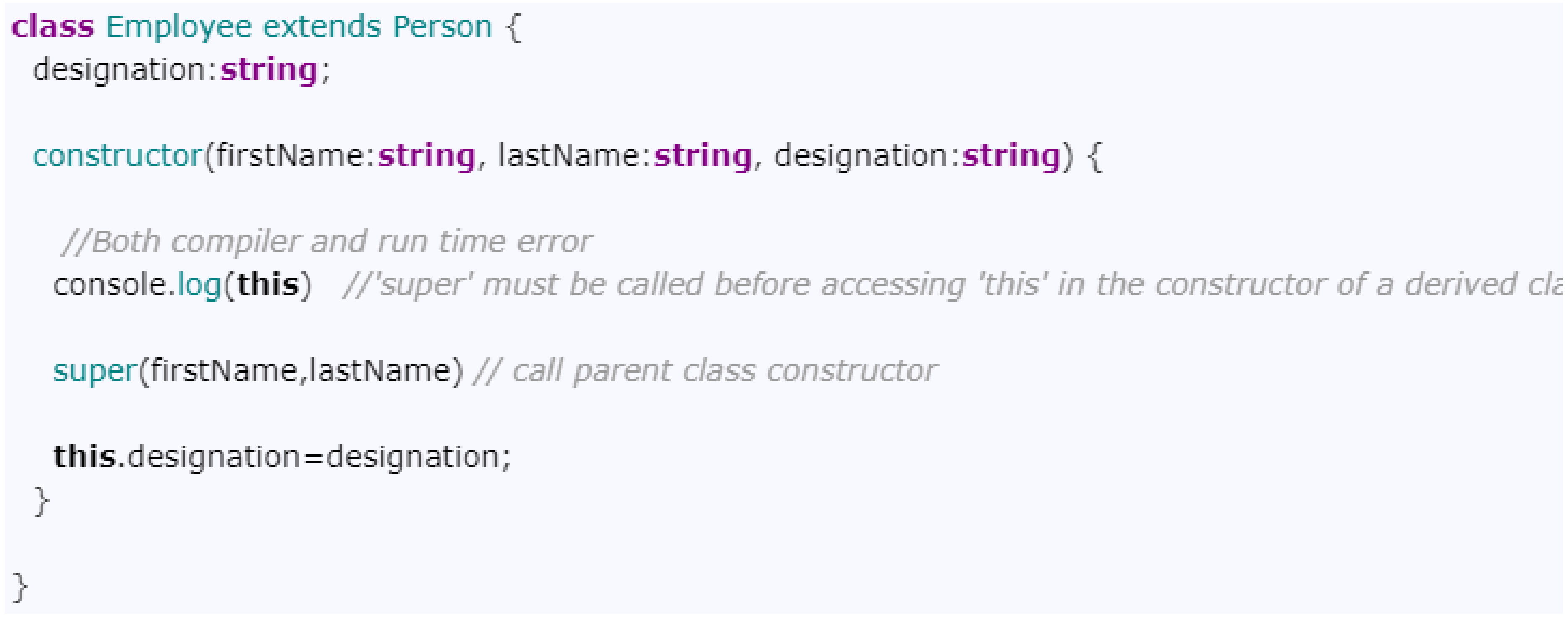
```
class Person {
  firstName:string;
  lastName:string;

  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;
  }

}

let e = new Employee("Jon","Snow","Manager")
console.log(e) //Employee: {   "firstName": "Jon",  "lastName": "Snow",   "designation": "Manager" }
```

Constructor functions in the derived class

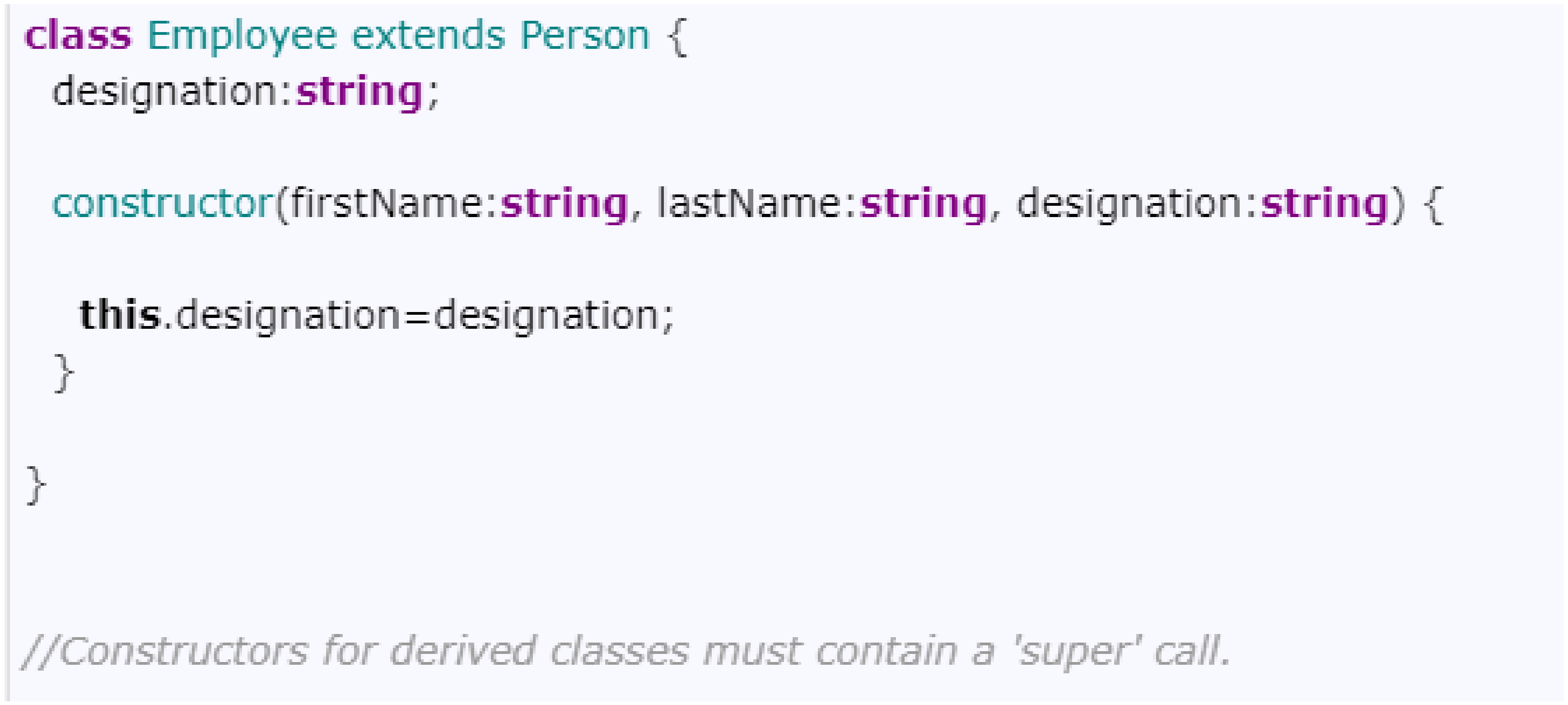
- Call the super method before accessing ‘this’ or before we return from the derived constructor. Trying to access the variable ‘this’ will result in both compile-time and run-time errors (“**‘super’ must be called before accessing ‘this’ in the constructor of a derived class**“).



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

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

    //Both compiler and run time error
    console.log(this)   //'super' must be called before accessing 'this' in the constructor of a derived cla

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

    this.designation=designation;
  }

}
```

Constructor functions in the derived class

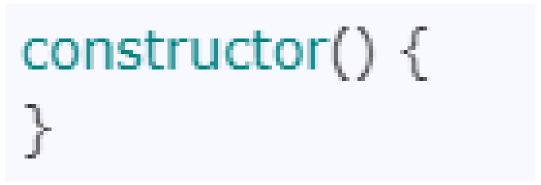
- Not invoking the super call in the derived class will also result in a compiler error “**Constructors for derived classes must contain a ‘super’ call**“



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

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

    this.designation=designation;
  }

}


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

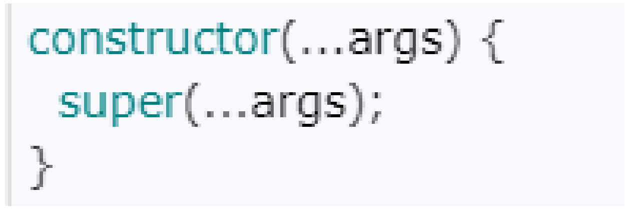
Class without constructor

- We can create a class without a constructor method. In such cases, **JavaScript** (not TypeScript) automatically uses a default constructor.
- The default constructor is an empty constructor.



```
constructor() {
}
```

- But if the class is a derived class then the default constructor is as shown below



```
constructor(...args) {
  super(...args);
}
```

Multiple Constructor methods

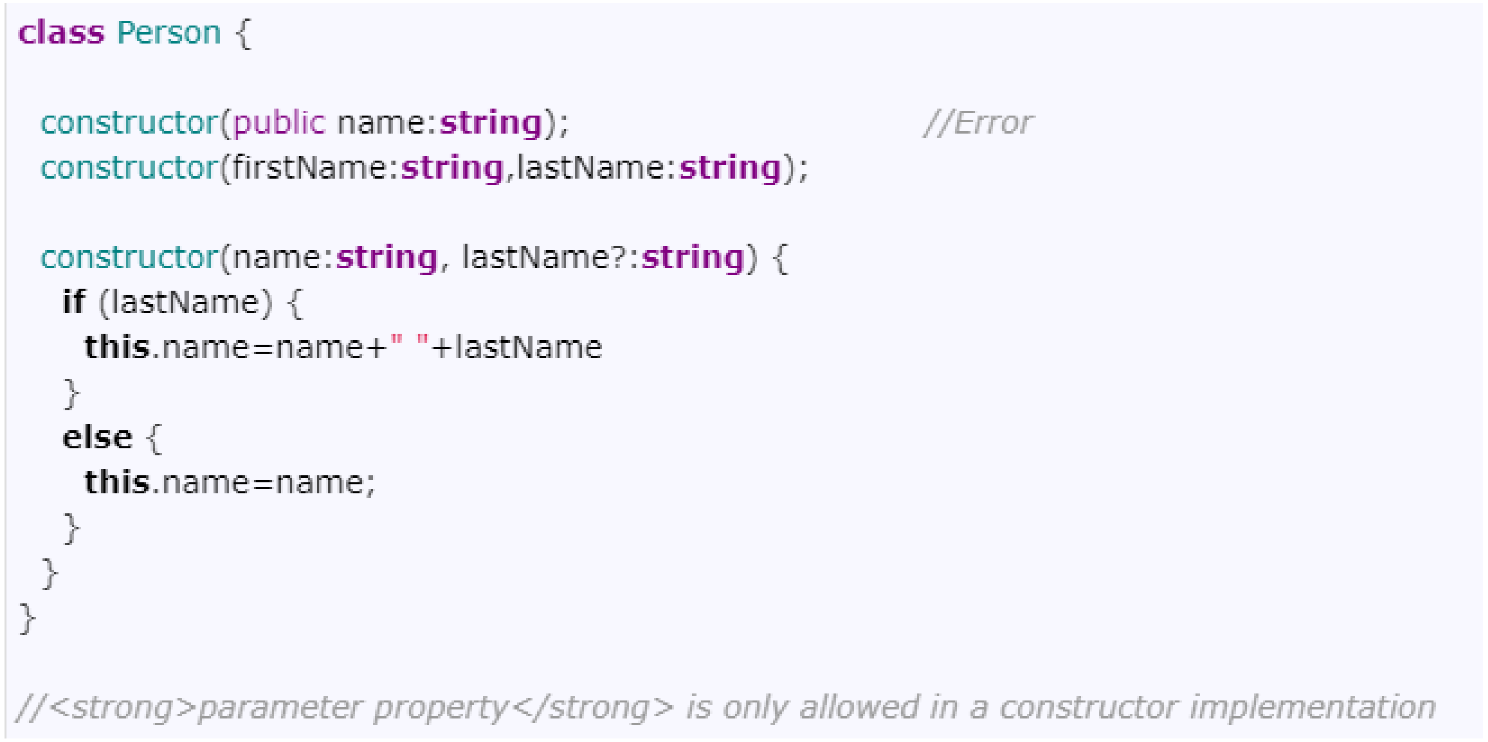
- A class can have **only one implementation** of the constructor method. Having more than one constructor function will result in an error.
- But we can take advantage of function overload to create multiple overload signatures of the constructor function.

Multiple Constructor methods

```
class Person {

  name:string;

  constructor(name:string);                          //constructor function signature 1
  constructor(firstName:string,lastName:string);     //constructor function signature  2

  constructor(name:string, lastName?:string) {       //actual constructor function
    if (lastName) {
      this.name=name+" "+lastName
    }
    else {
      this.name=name;
    }
  }
}

let p = new Person("Jon", "Snow");
console.log(p)   //Person: { "name": "Jon Snow" }

p = new Person("Samwell Tarly");
console.log(p)   //Person: {  "name": "Samwell Tarly" }
```

Multiple Constructor methods

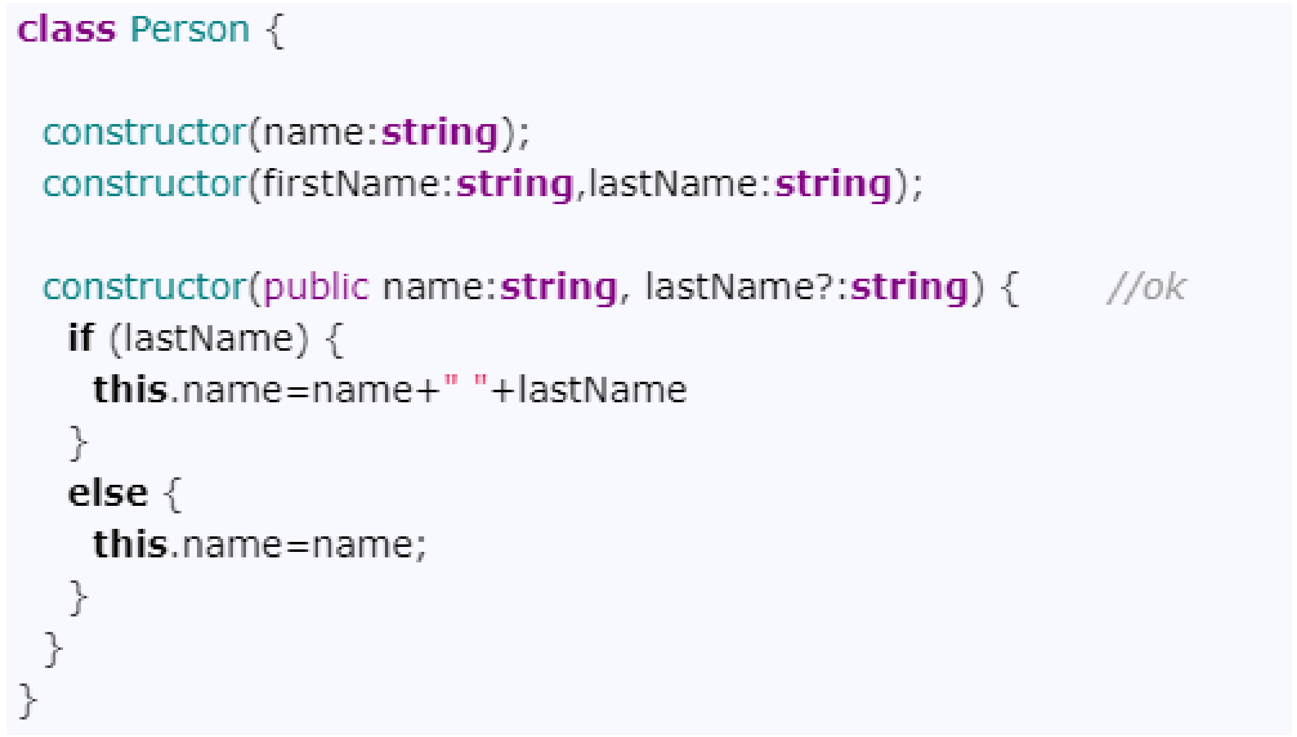
- Note that a **parameter property** is only allowed in a constructor implementation. For example, prefixing the public modifier to the name property in the constructor function signature results in a compiler error.



```
class Person {

  constructor(public name:string);                    //Error
  constructor(firstName:string,lastName:string);

  constructor(name:string, lastName?:string) {
    if (lastName) {
      this.name=name+" "+lastName
    }
    else {
      this.name=name;
    }
  }
}

//<strong>parameter property</strong> is only allowed in a constructor implementation
```

Multiple Constructor methods

- We can prefix the public modifier to the name property only in the constructor implementation function.



```
class Person {

  constructor(name:string);
  constructor(firstName:string,lastName:string);

  constructor(public name:string, lastName?:string) {     //ok
    if (lastName) {
      this.name=name+" "+lastName
    }
    else {
      this.name=name;
    }
  }
}
```