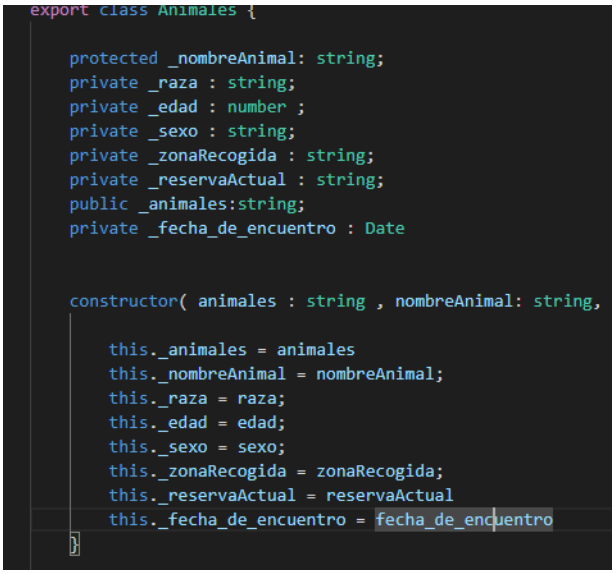
Proyecto clases con herencia
Lucas Coronilla Salmerón

Encapsulación:

Se denomina **encapsulamiento** al ocultamiento del estado, es decir, de los datos miembro de un objeto de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto.

Cada objeto está aislado del exterior, es un módulo natural, y la aplicación entera se reduce a un agregado o rompecabezas de objetos. El aislamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

Para manejar el encapsulamiento especificaremos nuestros datos de objetos como private y solo podremos leer sus datos mediante un método **get()** y otro método **set()**



```
export class Animales {

    protected _nombreAnimal: string;
    private _raza : string;
    private _edad : number ;
    private _sexo : string;
    private _zonaRecogida : string;
    private _reservaActual : string;
    public _animales:string;
    private _fecha_de_encuentro : Date


    constructor( animales : string , nombreAnimal: string,

        this._animales = animales
        this._nombreAnimal = nombreAnimal;
        this._raza = raza;
        this._edad = edad;
        this._sexo = sexo;
        this._zonaRecogida = zonaRecogida;
        this._reservaActual = reservaActual
        this._fecha_de_encuentro = fecha_de_encuentro
    }
```

```
    set Nombreanimal(value:string) {
        this._nombreAnimal = value
    }


    get Nombreanimal() {
        return  this._nombreAnimal
    }
```

Herencias:

Cuando hablamos de herencia en programación no nos referimos precisamente a que algún familiar lejano nos ha podido dejar una fortuna, ya nos gustaría. En realidad se trata de uno de los pilares fundamentales de la programación orientada a objetos. Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase.

En este proyecto tendremos una clase llamada animales y sus dos herencias que derivan de la clase principal la cual separará perros de caballos.

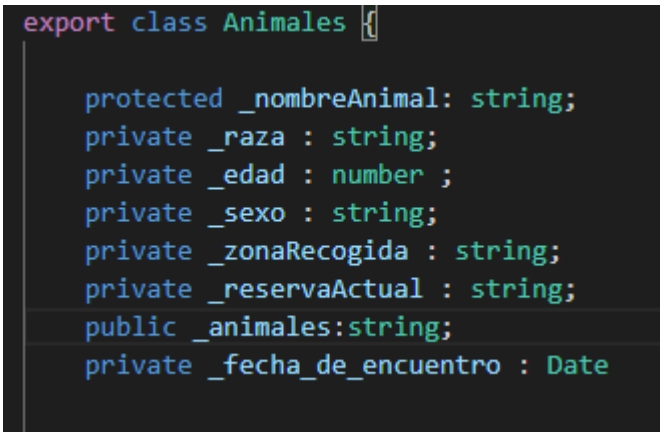
```
export class Animales {

    protected _nombreAnimal: string;
    private _raza : string;
    private _edad : number ;
    private _sexo : string;
    private _zonaRecogida : string;
    private _reservaActual : string;
    public _animales:string;
    private _fecha_de_encuentro : Date
```

```
import { Animales } from './Animales';
export class Caballo extends Animales {

  private _aptoMonta : string;
  private _aptoTiro:string;
```

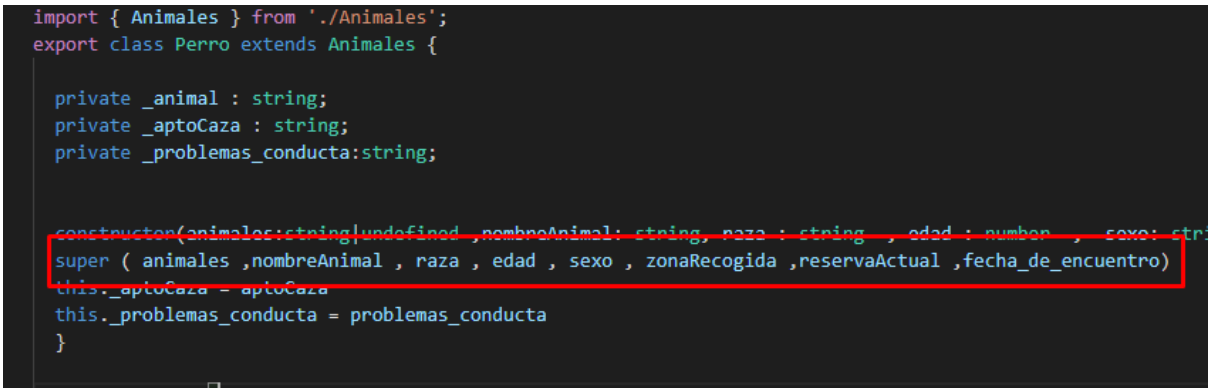
```
import { Animales } from './Animales';
export class Perro extends Animales {

  private _animal : string;
  private _aptoCaza : string;
  private _problemas_conducta:string;
```

Sobreescritura del constructor:

La idea detrás de la herencia es crear clases más complejas y posiblemente necesites más información para crear un objeto de una clase extendida, por eso utilizaremos el método **super()** para recoger el constructor de la clase principal y extenderlo en la subclase.



```
export class Animales {

    protected _nombreAnimal: string;
    private _raza : string;
    private _edad : number ;
    private _sexo : string;
    private _zonaRecogida : string;
    private _reservaActual : string;
    public _animales:string;
    private _fecha_de_encuentro : Date
```



```
import { Animales } from './Animales';
export class Perro extends Animales {

  private _animal : string;
  private _aptoCaza : string;
  private _problemas_conducta:string;


  constructor(animales:string|undefined ,nombreAnimal: string, raza : string , edad : number , sexo: str
  super ( animales ,nombreAnimal , raza , edad , sexo , zonaRecogida ,reservaActual ,fecha_de_encuentro)
  this._aptoCaza = aptoCaza
  this._problemas_conducta = problemas_conducta
  }
```

Polimorfismo:

El polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

```
switch (y){
    case 1:
     a.Nombreanimal = await leer("Introduce un nuevo nombre")
    break
    case 2:
     a.Raza = await leer ("introduce una nueva raza")
     break
    case 3:
     a.Edad = parseInt (await leer("introduce una nueva edad"))
     break
    case 4:
     a.Sexo = await leer("introduce un nuevo sexo")
     break
    case 5:
     a.ZonaRecogida = await leer("cambie el area donde fue recojido el animal")
     break
     case 6:
     a.ReservaActual= await leer("cambie el area de donde se hospeda el animal")
     break
     case 7:
     a.fechaDeEncuentro= new Date(await leer("Cambie la fecha donde fue encontrado el animal"))
     case 8:
        if (a instanceof Perro == true){
     perro.Aptocaza = await leer("cambie si el animal es apto para caza o no")
        }else{
     caballo.aptoMonta = await leer("cambie si el caballo es apto para la monta")
        }
     break
     case 9:
        if (a instanceof Perro == true){
        perro.Problemas_conducta = await leer("cambie si el animal tiene problemas de conducta")
                }else{
        caballo.Aptotiro = await leer("cambie si el caballo es apto para el tiro")
                }
        break
    case 10:
        o=0
        break
```

En esta función podemos observar como podemos recorrer el array y modificar datos de 2 clases distinta como son caballo y perro ya que el el array tiene clase padre de animales permitiéndonos modificar las 2 subclases.