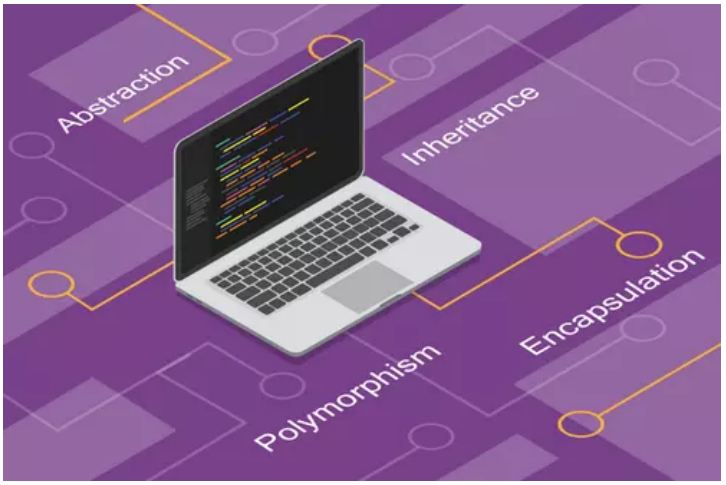
Adrià García Pons
2º ASIR | ASGBD

Proyecto Clases con Herencias



Índice

Adrià García Pons
2º ASIR | ASGBD

Introducción:

- **Introducción a la práctica.**

En esta práctica hemos profundizado en la programación orientada a objetos en Typescript. En concreto, hemos creado un **objeto principal, "ventilador",** y le hemos creado **dos herencias** al mismo; **"Ventilador de techo"** y **"Climatizador"**.

Hemos creado una serie de **métodos que nos permiten interactuar** y realizar pruebas **con estos objetos que hemos creado**.

Estos métodos los veremos más a fondo en cada uno de sus correspondientes apartados, pero sirven para funciones básicas como **listar** los ventiladores, **buscar** un ventilador, **crear**, **eliminar** y **editar** un ventilador, **insertar nuevos ventiladores aleatorios**, etc.

- **Herramientas que utilizaremos.**
  - **Visual Studio Code**: Será el entorno donde hemos realizado el grueso del proyecto, es decir, todas las funciones de nuestro typescript.
  - **Git y Github:** Por un lado, Git será la aplicación que utilizaremos para exportar el src de nuestro proyecto a Github, el cual es el sitio web donde tendremos nuestro repositorio de forma online.
  - **Typescript:** Descargaremos el paquete de Typescript para utilizarlo como compilador.
  - **NodeJS:** Será lo que nos permita llevar a cabo toda la configuración de paquetes y módulos en nuestro proyecto y por así decirlo ocupa un rol más administrativo sobre Typescript.

- **Elementos a tener en cuenta.**

Este proyecto lo considero una continuación del anterior por lo que no volveremos a ver:

- Creación de cuenta y de repositorio de Github (desde web).
- Profundización sobre los comandos de Git.
- Comandos básicos de Typescript.
- Subida del proyecto a Github mediante Git.

Adrià García Pons
2º ASIR | ASGBD

Práctica:

- **Explicación general de la práctica.**

En esta práctica como hemos dicho anteriormente hemos creado el objeto principal que es Ventilador, y sobre este le hemos creado dos herencias; Ventilador de techo y Climatizador.

He intentado atenerme lo más posible a la realidad por lo que los distintos campos de cada objeto son elementos distinguibles entre estos tipos de ventiladores en la realidad.

Ventilador de Sobremesa (Objeto principal)

Variables de este Ventilador (Que el resto heredan)

- **id**: Identificador único de los ventiladores
- **tipo**: El tipo de ventilador que es (Sobremesa /Techo/Climatizador)
- **marca**: La marca del mismo
- **potencia**: Variable numérica
- **color:** color del mismo
- **silenciador**

También heredan métodos los cuales luego podrán ser editados o no, los principales son:

- **Precio():** Calcula el precio del ventilador en base a la potencia, peso, y si tiene o no tiene silenciador.
- **Todo()**: Ofrece todas las características del ventilador.
- **General()**: Ofrece solo algunas características del ventilador.

- **Ventilador de Techo** (Objeto heredado de Ventilador Sobremesa)

Variables:
- Heredaría del método super: **Id/Tipo/Marca /Potencia/Color /Silenciador**
- **Aspas:** Variable en la que definiríamos el material de sus aspas.

Se modifican también también los métodos, de forma que el método de **precio()** tendría en cuenta el material de las aspas para los cálculos, y en el método **todo()** se mostraría la información también de las aspas.

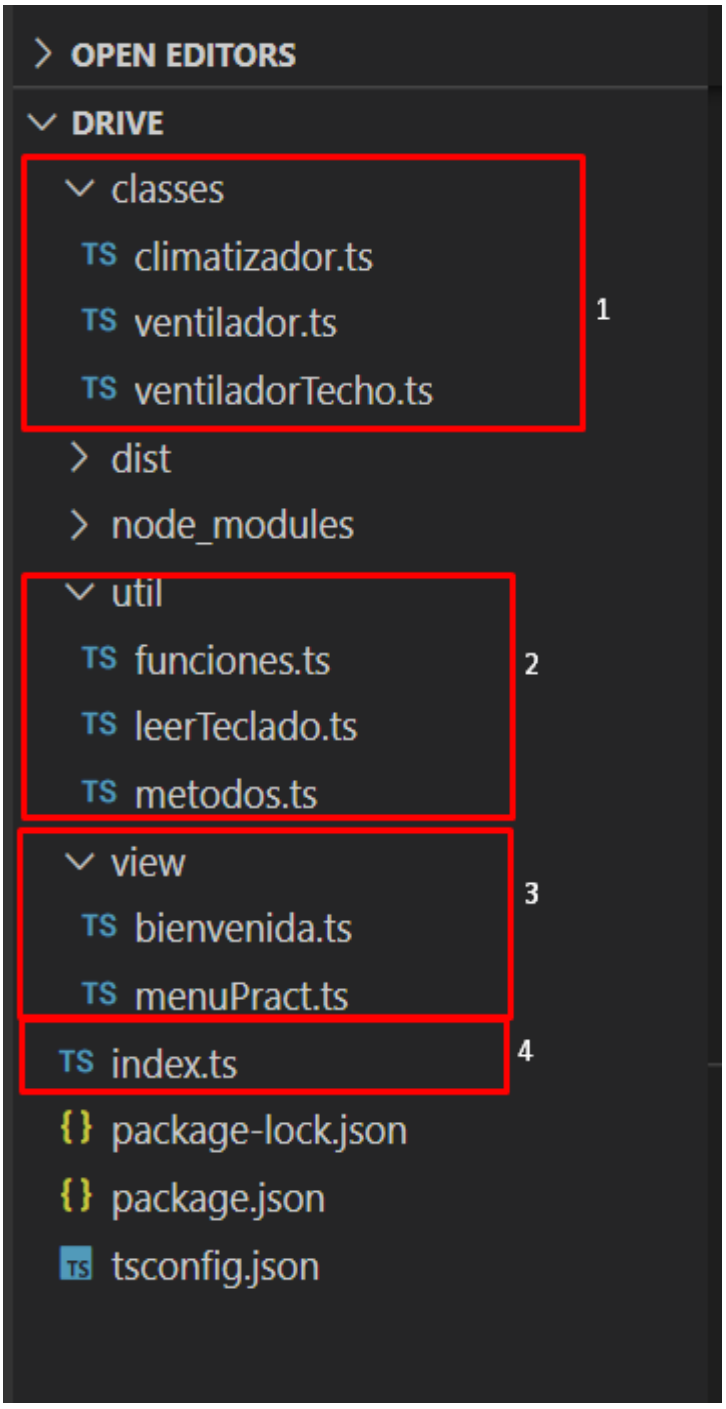
- **Climatizador** (Objeto heredado de Ventilador Sobremesa)

Variables:
- Heredaríamos del método super: **Id/Tipo/Marca/Potencia/ Color/Silenciador**
- **Cantidad Agua:** Casi todos los climatizadores guardan agua en su interior para que el aire que suelten tenga cierta humedad.

Se modifican también también los métodos: **precio()** tendría en cuenta la cantidad de agua, y en el método **todo()** se mostraría la información también la cantidad de agua.

- **Estructuración del código.**

Tenemos todo la práctica dividida en distintos directorios:



- **1: Classes**

En este directorio tenemos los 3 objetos de nuestra práctica, ventilador (principal), climatizador y ventilador de techo (herencias).

- **2: Util**

En este directorio tenemos el método de leerTeclado, un segundo fichero de funciones las cuales son las funciones que cargaremos directamente desde el menú de la práctica, y un tercero que son metodos, los cuale serán llamados desde las funciones para tareas concretas.

- **3: View**

El código de estos archivos son o bien puro console.log o código que solo queremos que se ejecute al iniciar el programa.

- **4: Index**

El corazón de nuestra práctica, desde la que llamaremos a las funciones.

- **Implementación del menú y funcionamiento del Index.**

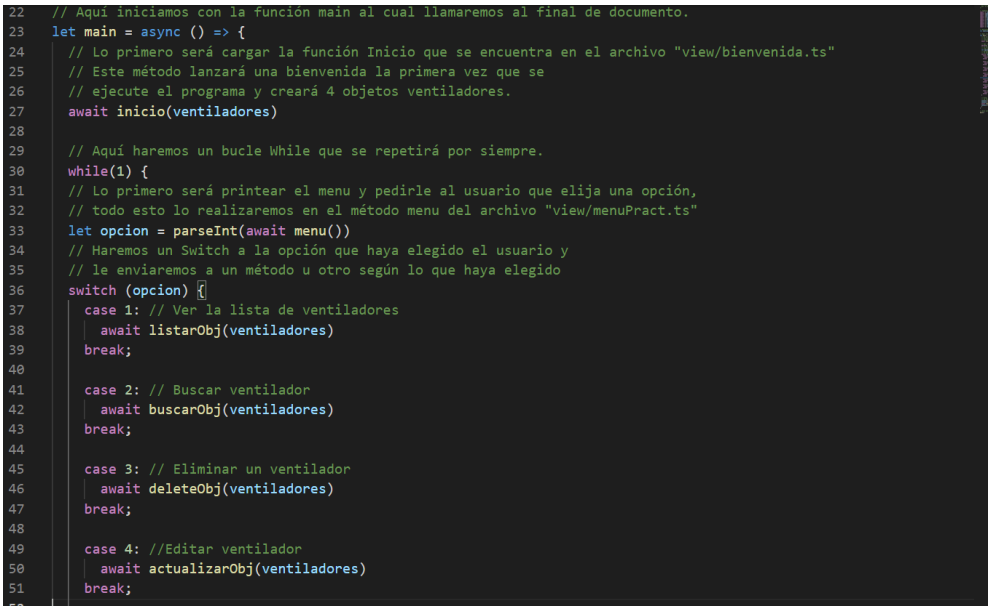
En el index apenas tenemos código, al ser el archivo que se ejecuta al iniciar la práctica es primordial siempre tenerlo lo más claro posible para no complicar la escalabilidad y la comprensión del proyecto.

En nuestro Index únicamente realizamos imports de los métodos que vamos a llamar desde el index (Que es el archivo de funciones del directorio util y los métodos del view).

Al iniciar el main por primera vez **declararemos** el Array Ventiladores, que será un **array de objetos** y llamaremos al método “inicio()” el cual se encuentra en /view/bienvenida.ts, y el cual simplemente realizará 6 imports de objetos de forma manual.

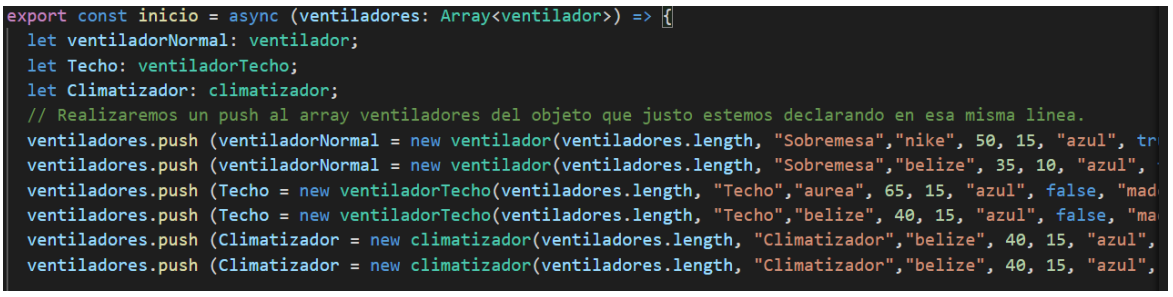
Tras la declaración del Array y de llamar al método Inicio, nos adentraremos en un bucle While el cual cargará de forma continua un menú gracias al método menu de /view/menuPract donde está escrito el menú.

Captura del index:



```
// Aquí iniciamos con la función main al cual llamaremos al final de documento.
let main = async () => {
  // Lo primero será cargar la función Inicio que se encuentra en el archivo "view/bienvenida.ts"
  // Este método lanzará una bienvenida la primera vez que se
  // ejecute el programa y creará 4 objetos ventiladores.
  await inicio(ventiladores)

  // Aquí haremos un bucle While que se repetirá por siempre.
  while(1) {
  // Lo primero será printear el menu y pedirle al usuario que elija una opción,
  // todo esto lo realizaremos en el método menu del archivo "view/menuPract.ts"
  let opcion = parseInt(await menu())
  // Haremos un Switch a la opción que haya elegido el usuario y
  // le enviaremos a un método u otro según lo que haya elegido
  switch (opcion) {
    case 1: // Ver la lista de ventiladores
      await listarObj(ventiladores)
    break;

    case 2: // Buscar ventilador
      await buscarObj(ventiladores)
    break;

    case 3: // Eliminar un ventilador
      await deleteObj(ventiladores)
    break;

    case 4: //Editar ventilador
      await actualizarObj(ventiladores)
    break;
```

Realización de los imports:



```
export const inicio = async (ventiladores: Array<ventilador>) => {
  let ventiladorNormal: ventilador;
  let Techo: ventiladorTecho;
  let Climatizador: climatizador;
  // Realizaremos un push al array ventiladores del objeto que justo estemos declarando en esa misma linea.
  ventiladores.push (ventiladorNormal = new ventilador(ventiladores.length, "Sobremesa","nike", 50, 15, "azul", tr
  ventiladores.push (ventiladorNormal = new ventilador(ventiladores.length, "Sobremesa","belize", 35, 10, "azul",
  ventiladores.push (Techo = new ventiladorTecho(ventiladores.length, "Techo","aurea", 65, 15, "azul", false, "mad
  ventiladores.push (Techo = new ventiladorTecho(ventiladores.length, "Techo","belize", 40, 15, "azul", false, "ma
  ventiladores.push (Climatizador = new climatizador(ventiladores.length, "Climatizador","belize", 40, 15, "azul",
  ventiladores.push (Climatizador = new climatizador(ventiladores.length, "Climatizador","belize", 40, 15, "azul",
```

- **Explicación de cada ejercicio.**
    - **Métodos llamados por las funciones:**

Estos métodos están a parte de las funciones ya que las funciones son llamadas una única vez y por el index, sin embargo, estos métodos los utilizaremos varias veces siendo llamados por varias funciones.

Estos métodos nos permiten ahorrar escribir el código de forma duplicada, fomentan la escalabilidad de la práctica y por supuesto hacen que esta sea mucho más comprensible.

Trabajan en un segundo plano ya que no son llamados por el usuario directamente (Podrían ser llamados por el usuario pero no es el caso en esta práctica).

- **Método verObjId**

Este método recibirá como **parámetro un array** (Ya que el proyecto solo trabaja con una array será ventiladores), **un identificador**, que podrá ser string o entero, y un string que concretará el **tipo de búsqueda** a realizar.

Este método recorre el Array que le proporcionemos y dependiendo del tipo de búsqueda seleccionará a los objetos de una u otra manera.

En caso de buscar por Id seleccionará el objeto cuyo campo identificador sea el mismo al que hayamos introducido.

En caso de buscar por marca o por tipo de objeto irá imprimiendo en pantalla todo objeto que tenga la marca o sea del tipo de objeto que hayamos introducido.

Este método es llamado por funciones como Crear, Eliminar, Editar por lo que nos ahorra mucho código el crear este método independiente.

- **Método getNewId**

Como ya sabemos, nuestros ventiladores tienen todos un identificador único el cual es un número, sin embargo, para evitar duplicidades del Id y que el Id sea siempre el más pequeño posible (para ordenar mejor el Array de objetos), se generará automáticamente el Id cada vez que queramos crear un nuevo ventilador.

Este método asigna la Id al objeto en función de las posiciones libres del array. Siempre que se crea un nuevo objeto se realizará un push al array por lo que físicamente en el array siempre estará al final.

Este método recibirá como parámetro un Array de objetos (Ventiladores).
Lo primero que hace el método es asignar por defecto la Id que devolverá al tamaño del array, ya que el tamaño del Array siempre va a ser la posición del último elemento +1, con esto nos aseguramos que siempre vaya a proporcionar la Id más grande en caso de que no encuentre ninguna menor libre.

El método recorrerá el Array e irá comprobando que la variable Identify, que es la que explicamos justo antes, sea mayor que el índice dentro del Array del objeto actual y por supuesto que el identificador del propio objeto sea mayor que el índice de este dentro del array.

En caso de que ambas condiciones se cumplan se le asignará a Identify un nuevo valor, en concreto, será la diferencia que exista entre el identificador del objeto y su posición en el array la que nos revele el Id a utilizar (en caso de que no haya hueco libre siempre esta diferencia será de 0).

Tras esto, devolverá el Identify, es decir, el Id para el objeto a la función que le hubiera llamado.

Este método es llamado tanto por la función de crear nuevo objeto como por la función de crear nuevos objetos aleatorios.

- **Método ordenarArray**

Este método es complementario al método getNewId y soluciona el gran problema que este tiene, el método getNewId detecta que hay un hueco libre entre las Id's mediante la comparación de la posición del objeto dentro del array con el campo identificador del propio ejemplo. Sin embargo, cuando realizamos el push del objeto al array este siempre es introducido al final de esta, por lo que existirán grandes diferencias siempre entre la posición del objeto en el array y su campo identificador. Este método simplemente soluciona este problema.

El método ordenarArray como su nombre indica simplemente recorre todo el Array (de forma doble), comparando cada elemento con cada elemento e intercambiandolos de lugar en caso de que uno con mayor indice ocupe una menor posición dentro del Array.

Este método es llamado por la función de Crear Nuevo Objeto pero su funcionalidad al ser tan distinta a la de crear un nuevo objeto considero que debe ir como método independiente.

- **Método leerTeclado**

Este método nos lo dio el profesor y su única funcionalidad es facilitarnos la entrada de datos.

Este método lo considero bastante cómodo ya que nos permite en una misma línea solicitar datos e imprimir por pantalla el tipo de dato que queremos recopilar.

- **Funciones llamadas por el index:**

Estas funciones como su nombre indica recogen el gran peso de nuestra aplicación, funcionan todas de forma independiente pero son llamadas unicamente por el índice. A su vez, casi todas de estas funciones dependen de los métodos citados anteriormente para funcionar.

- **Función listar objetos**

Esta función simplemente recorrerá el array, llamará al método del objeto “general()” el cual imprimirá los valores generales de cada objeto, y también a la función “precio()” que irá calculando e imprimiendo el precio de cada objeto.

- **Buscar Objeto**

En esta función comenzaremos pidiendo al usuario que nos inserte el criterio de búsqueda el cual podrá ser Id, Marca o Tipo de objeto.

En función del criterio de búsqueda dará cierta información (Si es marca dará una lista de las marcas conocidas y si es tipo dará una lista de los tipos de objetos existentes), tras esto, solicitará al usuario que inserte la información a buscar.

Ahora solo enviará al método verObjId() el Array en cuestión (ventiladores), el filtro de búsqueda (Marca/Id/Tipo), y la información a buscar. El método verObjId como vimos antes nos mostrará toda la información de cada uno de los objetos que cumplan los criterios mediante el método todo()

- **Eliminar Objeto**

Comenzaremos pidiendo al usuario que inserte el Id del objeto a eliminar, tras que el usuario lo inserte realizaremos una búsqueda en el Array de ventiladores y una vez lo encontremos se lo mostraremos en pantalla y pediremos que confirme el eliminado del objeto mediante el método verObjId(), en caso de que confirme le realizaremos un splice al Array.

- **Editar Objeto**

Como antes pediremos al usuario que nos indique el Id del objeto a editar y tras esto comenzaremos a buscarlo dentro del array, una vez encontrado, tal como antes, lo mostraremos en pantalla mediante el método verObjId() y pediremos confirmación al usuario.

Si el usuario confirma comenzaremos con la edición de cada uno de los campos donde se le presentarán al usuario el valor original de los campos y para campos como marcas o aspas las distintas opciones que tiene para elegir.

Finalmente, editaremos el Array mediante un map.

- **Nuevo Objeto**

Comenzaremos generando un Id con el método getNewId(), tras esto, pediremos al usuario que nos indique el tipo de ventilador a crear, si de sobremesa, de techo, climatizador…

Una vez nos lo indique el usuario iremos pidiendo que nos inserte la información de los campos uno a uno, y tras que haya insertados todos haremos un push al nuevo objeto creado, tras esto, le mostraremos por pantalla el objeto resultante mediante verObjId()

- **Ver campos de Objeto**

Esta función unicamente pedirá al usuario que elija el tipo de ventilador del cual quiere ver los campos (Sobremesa/techo/climatizador), y una vez lo introduzca recorreremos el array hasta que encontremos el primer objeto que sea del tipo introducido.

Finalmente mostraremos los campos de este objeto encontrado.

- **randomGenerater**

Esta función pedirá al usuario que introduzca el número de objetos a crear, y tras esto el tipo que quiera que sean estos objetos(Sobremesa/Techo/ Climatizador/Cualquiera). Tras esto, comenzará a generar los objetos de uno en uno asignándoles el tipo (En caso de cualquiera le asignará uno de los otros 3), y a cada una de sus variables les asignará un valor de forma aleatoria.

- **Explicaciones teóricas de objetos**
  - **Creación de clase y de método constructor**

Para crear la clase la crearemos mediante “export class <nombre>” y le estableceremos todas las variables y el tipo de variable del objeto.

Luego el constructor lo crearemos simplemente poniendo “constructor”, indicando los parámetros que recibirá que serán básicamente las variables necesarias para construir nuestro objeto y finalmente realizaremos los set pertinentes.

```
export class ventilador {
  protected id: number;
  protected tipo: String;
  protected potencia: number; // para acceder en la subclase
  protected peso: number; // para acceder en la subclase
  protected color: String; // para acceder en la subclase
  protected marca: String; // para acceder en la subclase
  protected silenciador: boolean; // para acceder en la subclase

  constructor(id: number, tipo: string, marca: string, potencia: number, peso: number, color: string, silenciador:
    this.id= id;
    this.tipo = tipo;
    this.marca = marca;
    this.potencia = potencia;
    this.peso = peso;
    this.color = color;
    this.silenciador = silenciador;
  }
```

- **Creación de métodos**

Para crear un método de un objeto simplemente dentro de donde declaramos el objeto declararemos el propio método. En este caso vemos como declaramos precio el cual retornará una variable de tipo number y otros dos métodos los cuales devolverán variables de nuestro objeto.

```
  precio(): number {
    let precio = this.potencia/5 + this.peso/10;
    if (this.silenciador == true) {
      precio += 20;
    }
    return precio;
  }

  todo() {
    return `Id: ${this.id}, Tipo: ${this.tipo}, Marca: ${this.marca}, Potencia: ${this.potencia}, Peso: ${this.pes
  }
  general() {
    return `Id: ${this.id}, Tipo: ${this.tipo}, Marca: ${this.marca}, Potencia: ${this.potencia}`;
  }
}
```

- **Utilización de get y set**

Aquí podemos ver como podemos declarar un método get para que nos retorne el valor de una variable del objeto o un método set para modificar este valor, pidiendo como parámetro el dicho valor.

```
get aspasGet() {
  return this.aspas;
}

setAspas(aspas: string) {
  this.aspas=aspas
}
```

- **Creación de herencias, sobreescritura del constructor y uso de super**

En la segunda línea vemos como declararemos un objeto como hicimos con el anterior objeto sin embargo este al tratarse de una herencia tendremos que acompañar a esta declaración de “extends <objeto padre>”.

Como podemos ver unicamente le declararemos la variable aspas ya que el resto las hereda.

En la línea 4 declararemos el método constructor con sus parámetros igual que antes pero incluyendo la nueva variable de la herencia.

Abajo, eso si, realizaremos la función super() para decir que actualizaremos el valor de las variables indicadas tal y como se actualizan en el constructor del objeto padre.

```
1  import { ventilador } from './ventilador';
2  export class ventiladorTecho extends ventilador {
3    protected aspas: string;
4    constructor(id: number, tipo: string, marca: string, potencia: number, peso: number, color: string, silenciador:
5      super(id, tipo, marca, potencia, peso, color, silenciador);
6      this.aspas = aspas;
7    }
```

- **Sobreescritura de métodos**

Podemos ver como el método de Precio() lo sobreescribimos haciendo que la variable que tenemos en esta herencia entre en juego a la hora de calcular el precio del objeto.

```
precio(): number {
  let precio: number;
  precio = super.precio()
  interface AspasList {
    [key: string]: number;
  }
  let foo:AspasList = {};
  foo['madera'] = 15;
  foo['acero'] = 25;
  foo['plastico'] = 5;
  if(this.aspas=="madera" || this.aspas=="acero" || this.aspas=="plastico") {
    precio += foo[this.aspas];
  }
  return precio;
}
```

- **Polimorfismo**

Un claro ejemplo de polimorfismo lo tenemos en el método general() el cual es declarado en el objeto padre ventilador y no es sobreescrito por ninguna de las herencias, y a pesar de esto el método es compatible con cada uno de nuestros objetos ya que todos ellos son capaces de responder a lo que pide.

```
  }
  general() {
    return `Id: ${this.id}, Tipo: ${this.tipo}, Marca: ${this.marca}, Potencia: ${this.potencia}`;
  }
}
```

- **Encapsulación**

La encapsulación se refiere a ser capaces de gestionar correctamente nuestras variables para impedir que estas se puedan ver editadas sin nosotros quererlos por otras funciones.

Para proteger estas variables las declararemos como “protected”, mostraremos el valor de la misma mediante una función get concretada en el objeto y cambiaremos su valor mediante una función set en el objeto también creada.

```
protected potencia: number; // para acceder en la subclase
```

```
get potenciaObjeto()  {
  return this.potencia;
}
```

```
setPotencia(potencia: number) {
  this.potencia=potencia
}
```

```
key.setPotencia(potencia);
```