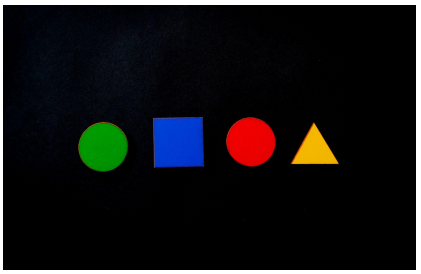
Clase II
OBJETOS



IEEE ITBA
STUDENT BRANCH

Índice

Arreglos

Introducción

Un arreglo es una estructura de datos que almacena una colección de valores de algún tipo (number, string, etc). Esto hace que se puedan crear y manipular listas de datos.

Imaginemos que tenemos una caja grande con muchas secciones dentro. Cada sección tiene un número identificatorio escrito. Estos números estarán dispuestos del 1 a N, siendo N la cantidad de secciones.
A lo que esté dentro cada sección los vamos a llamar “elemento”, y a cada número lo llamaremos índice. Si queremos obtener un elemento en específico, la forma más fácil de acceder a él es a través del índice mencionado. Sabemos que en ese índice va a estar el elemento que buscamos. Un arreglo es como esa caja, solo que los índices van del 0 a N-1.

Supongamos que tenemos un arreglo de N elementos, es decir, de N valores de cierto tipo. Una representación visual de un arreglo se podría hacer de esta forma:

Índice 0	Elemento 1
Índice 1	Elemento 2
...	
Índice N-1	Elemento N

Ejemplo práctico: Días de la semana

Continuando con el ejercicio anterior, de los días de la semana:

> *Hacer un programa que me imprima todos los días de la semana*

Ahora, nuestro problema puede ser resuelto más fácilmente. Esto se podría hacer mediante un arreglo de días de la semana.

```
let diasDeSemana = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes',
'Sábado', 'Domingo'];
```

En el mismo, cada posición representa un dato. Para obtener el valor *Lunes*, por ejemplo, habría que acceder a la primera posición del arreglo de días de la semana. La

primera posición está representada por el índice 0 del mismo. Entonces, para imprimir el valor *Lunes*, tendríamos que indicar:

```
console.log(diasDeSemana[0]);
```

Para agregar elementos al arreglo, se puede utilizar un método, llamado *push*. Para eliminar al último elemento de los arreglos, una buena forma sería con *pop*. Este método también permite obtener al elemento eliminado:

```
let diasDeSemana = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes',
'Sábado', 'Domingo'];

diasDeSemana.push('Octes');
console.log(diasDeSemana) // ['Lunes', 'Martes', 'Miércoles', 'Jueves',
'Viernes', 'Sábado', 'Domingo', 'Octes']

console.log(diasDeSemana.push()) // 'Octes'

console.log(diasDeSemana) // ['Lunes', 'Martes', 'Miércoles', 'Jueves',
'Viernes', 'Sábado', 'Domingo']
```

Para poder definir un arreglo a llenar en el futuro, también se podría definir al mismo como un arreglo vacío:

```
let diasDeSemana = [];

diasDeSemana.push('Lunes');
diasDeSemana.push('Martes');
.
.
.
diasDeSemana.push('Domingo');

console.log(diasDeSemana) // ['Lunes', 'Martes', 'Miércoles', 'Jueves',
'Viernes', 'Sábado', 'Domingo']
```

Resolución de ejercicio

Para imprimir todos los días de la semana del arreglo podríamos hacer un ciclo, como hicimos previamente. Necesitamos la longitud del arreglo, *diasDeSemana.length*, como condición para nuestro contador:

```
let diasDeSemana = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes',
'Sábado', 'Domingo'];

for(let i = 0; i < diasDeSemana.length; i++){
    console.log(diasDeSemana[i]);
}
```

Sin embargo, esa no es la única forma de recorrer un arreglo. Hay otras dos más que vamos a ver ahora, las cuales son 'for.. of' y 'for... in'. 'for... of' se utiliza para recorrer los elementos del arreglo (colección u objeto iterable), sin necesidad de obtener el índice:

```
for(let diaDeSemana of diasDeSemana){
    console.log(diaDeSemana);
}
```

Por último, tenemos la instrucción 'for... in', la cual se utiliza para recorrer todas las propiedades de un objeto. En este caso, nuestro objeto sería un arreglo, por lo tanto sus propiedades son los índices:

```
for(let i in diasDeSemana){
    console.log(diasDeSemana[i]);
}
```

El 'for... in' no se suele utilizar mucho con arreglos. No tiene un sentido predefinido y se puede romper el arreglo, sin contar que hay mejores formas de recorrer arreglos. Esto lo veremos más adelante.

Nuestra primera función

Introducción

Las funciones son bloques de código especiales que tienen la característica de aceptar un conjunto de datos especificado, o *inputs*,

procesarlo y devolver otro dato, u *output*. Algunas funciones ya vienen predefinidas en algunos lenguajes y otras son incorporadas en las distintas librerías disponibles.

Son útiles para evitar reescribir líneas de código en la realización de tareas similares y permiten dividir un problema en varias subtareas. No solo mejoran la legibilidad y prolijidad del código, sino que también permiten anticipar y encontrar errores de forma más sencilla.

Ejemplo de función

Teniendo el próximo problema:

> *Dada una variable de entrada arregloNumeros, que es un arreglo de números, devolver la suma de los mismos.*

Tenemos un arreglo de números. Ahora, queremos llamar a una *función* que haga lo que pedimos antes. Para declarar una función en TypeScript, lo recomendable es indicar el tipo de los datos que recibe y el tipo de dato de que devuelve. Con los tipos de datos que recibe, solo hace falta indicar el nombre de la variable y su tipo, como antes. Con el tipo de datos que devuelve, se pone después de indicar los argumentos que recibe la función:

```
function miFuncion(arregloNumeros: number[]): number { // Función que
recibe un arreglo de números y devuelve un número
      //..
}
```

También, existe otra manera de declarar una función, que es similar a la declaración de cualquier variable. Un ejemplo se muestra a continuación:

```
const miFuncion = (arregloNumeros: number[]): number => {
      //..
}
```

Lo que tenemos que hacer en este caso sería una función que reciba un arreglo de números, y nos devuelva otro número. El nombre de esta función lo elegimos de acorde a lo que hace, que es asignar devolver la suma de los números del arreglo. Podría ser algo como *sumaArreglo*.

```
function sumaArreglo(arregloNumeros: number[]): number{
```

```
    let suma = 0;
    for(let i = 0; i < arregloNumeros.length; i++){
      suma += arregloNumeros[i]
    }
    return suma;
}
```

Podríamos probar esta función mediante un ejemplo:

```
function sumaArreglo(arregloNumeros: number[]): number{
    let suma = 0;
    for(let i = 0; i < arregloNumeros.length; i++){
      suma += arregloNumeros[i]
    }
    return suma;
}

let numeros= [1, 5, 1, 5, 1, 5, 1, 5, 1, 5];
console.log(sumaArreglo(numeros)); // 30
```

Completando el programa

Inclusive, otra forma de emprolijar aún más el código, podría ser el colocar la porción que no quedó dentro de una función, dentro de otra función llamada 'main'. Finalmente, llamamos la función *sumaArreglo*desde el cuerpo de 'main':

```
function sumaArreglo(arregloNumeros: number[]): number{
    let suma = 0;
    for(let i = 0; i < arregloNumeros.length; i++){
      suma += arregloNumeros[i]
    }
    return suma;
}

function main(): void{
    let numeros= [1, 5, 1, 5, 1, 5, 1, 5, 1, 5];
    console.log(sumaArreglo(numeros)); // 30
}

main();
```

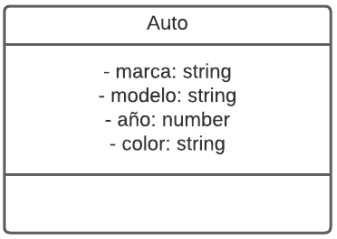
Para probar que lo que codeamos esté funcionando correctamente, podemos ejecutar ambas versiones y fijarnos si devuelven lo mismo.

¿Qué es un objeto?

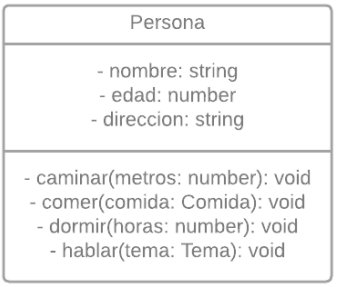
Hoy vamos a adentrarnos en el mundo de los objetos en un ambiente de programación. Algunos se podrían preguntar qué son estos objetos, y es una pregunta perfectamente válida. La respuesta viene atada a la representación de un objeto en la vida real.

En la Programación Orientada a Objetos, un objeto es la representación de una entidad o concepto del mundo real, en código.

Podemos definir un objeto que se llame Auto, y que represente una cantidad finita de propiedades del mismo, tales como marca, modelo, año y color. Cada propiedad tiene sus propios valores específicos.



A los objetos también se le pueden asignar métodos, o funciones, que son formas de manipular esos atributos. En el caso de que estemos modelando una Persona, podemos definir como propiedades su nombre, edad, dirección, y también métodos como caminar(), hablar(), comer(), dormir().



Tipos especiales de dato

En primer lugar, hay algunos tipos de datos que son genéricos o que quieren representar a la nada o lo desconocido. Hablaremos de *any*, *unknown*, *undefined*, *null*, y *void*. Algunos de ellos los vimos la clase anterior, pero no los habíamos explicado.

Tipo de datos *any*

A veces, cuando trabajamos con librerías externas, o queremos trabajar con una variable que acepte múltiples tipos de datos, no sabemos con qué nos podemos encontrar, o no podemos englobar a los objetos o tipos de datos.
En esos casos, podemos usar el tipo de dato que quiere decir "este objeto o tipo de datos no tiene validaciones". Se llama *any*, y es muy poderoso, por lo que debemos usarlo con cuidado y no hay que abusar de él, ya que puede causar muchos errores de ejecución no deseados.
Es una forma de invalidar cualquier especificidad y decir que una variable "puede ser cualquier cosa", literalmente.

Demos un vistazo a algunos usos de *any*:

```
let value: any;

value = true; //Ok
value = 35; // Ok
value = "Heisenberg"; // Ok
value = []; // Ok
value = {}; // Ok
value = Math.random; // Ok
value = null; // Ok
value = undefined; // Ok
value = new TypeError() // Ok
value = Symbol("type") // Ok

value.foo.bar // Ok
value.trim() // Ok
value() // Ok
new value() // Ok
value[0][1] // Ok
```

Aquí, podemos notar que ni nuestro IDE ni nuestro compilador nos va a avisar que hay errores o advertencias.

Los errores que puedan ocurrir al usar value como una función o querer acceder a propiedades inexistentes van a aparecer recién al momento de ejecutar el código,

cuando ya sabíamos de antemano que no funcionaba.
Lo mejor que se puede hacer antes de usar *any*, es definir tipos explícitos de datos para las variables y valores que se usen, siempre que sea posible.

Tipo de datos *unknown*

También tenemos el tipo de datos *unknown* que, a diferencia de *any*, asume que la variable puede ser "o todo o nada". Por lo general, se utiliza cuando no se sabe con qué tipo de variable se está trabajando, para obligar al compilador a que haya validaciones o chequeos previos a usar las propiedades de las variables con unknown:

```
let value: unknown;

let value1: unknown = value; // Ok
let value2: any = value; // Ok
let value3: boolean = value; // Error
let value4: number = value; // Error
let value5: string = value; // Error
let value6: object = value; // Error
let value7: any[] = value; // Error
let value8: Function = value; // Error

value.foo.bar // Error
value.trim() // Error
value() // Error
new value() // Error
value[0][1] // Error

function manejarUnknown(valor: unknown)
{
  if (typeof valor=== "string") {
    // valor está ahora acotado al tipo
'string'
    console.log(valor.toUpperCase());
  } else if (valor instanceof Date) {
    // valor está ahora acotado al tipo
'Date'
    console.log(valor.getFullYear());
  } else {
    // valor es todavía del tipo
'unknown'
    console.log("Tipo desconocido:",
valor);
  }
} // Esta función valida el tipo de
datos antes de hacer uso de las
```

```
propiedades y métodos específicos para
cada tipo de datos. Ese sería el uso
correcto de unknown
```

Ahora que tenemos formas de definir el “todo”, vamos a descubrir una manera de describir la “nada”, o “error”, en algunos casos.

Tipo de datos *undefined*

Supongamos que tenemos una variable del tipo *number*, pero no le asignamos ningún valor:

```
let queNumeroEsEste: number;
```

¿Qué valor puede tener después de esa línea *queNumeroEsEste*? La respuesta es “ninguno, ya que nunca se le puso un valor”. Para este caso, el valor de esta variable será indefinido (*undefined*). Eso quiere indicar que, si bien la variable es un número, todavía no tiene un valor asignado o, mejor dicho, ¡no existe!

Ahora bien, cuando uno quiera imprimir esta variable, para validar que así sea, nos pasará que arrojará un error de compilación. Esto es porque el tipo *undefined*, no coincide con el tipo *number*, .

Probemos, ejecutando el siguiente código:

```
let queNumeroEsEste: number;

console.log(queNumeroEsEste); // Error
```

Si queremos ver esta variable *undefined*, tendríamos que hacer un tipado “multiple”. Esto es, decir que la variable *queNumeroEsEste* puede ser 2 tipos de dato. *number* o *undefined*. Veamos las siguientes líneas de códigoÑ

```
let queNumeroEsEste: number | undefined; // let queNumeroEsEste?: number es
su equivalente

console.log(queNumeroEsEste); // undefined
```

Tipo de datos *null*

También, el tipo *null* cumple una función similar. Este último suele indicar la ausencia intencional de valor, o que está vacío. Por lo general, uno debería tener la posibilidad de asignarle a un objeto el valor nulo, mientras que no debería poder asignarle a un objeto el valor “no está definido”.

```
let queNumeroEsEste : number | undefined = undefined; // Esto es normal y no tendria sentido
let queNumeroEsEste: number | null = null; // Esto es normal y tiene sentido
```

Una forma de representar al vacío es utilizar el tipo *void*, que no quiere indicar ni *undefined* ni *null*. Este tipo se utiliza especialmente para indicar que una función no retorna algo. Funciones de este estilo podrían ser funciones que solo imprimen en pantalla, o cambian la condición de un objeto. Es coherente en cuanto a la declaración de variables, ya que toma *null* y *undefined* como únicos valores posibles.

Enums

Qué son los *enum* y cómo se utilizan

Los Enums son una forma de modelar diferentes tipos de estado de una entidad, como por ejemplo, los diferentes colores posibles. Para ello, se utilizan estructuras de datos que contienen distintas constantes con nombre, las cuales pueden ser asignadas valores de 3 formas: por defecto, inicializadas y completamente inicializadas.

Tres formas de inicializar un enum

En la asignación por defecto, la primera constante tiene el valor 0, y las siguientes tienen el valor anterior + 1. Por lo general, este tipo de inicialización se utiliza si no nos interesa realmente el valor, sino que el nombre de la constante. Un ejemplo de ello podría ser:

```
enum Frutas {
  Manzana, // 0
  Naranja, // 1
  Banano,  // 2
  Mango,   // 3
  Uva      // 4
}
```

En la asignación por inicialización, se indica el primer valor numérico del enum, y los demás valores se inicializan automáticamente en el anterior + 1. Por ejemplo:

```
enum Numeros {
  Diez = 10,
  Once,   // 11
  Doce,   // 12
  Trece,  // 13
  Catorce // 14
}
```

Por último, en la asignación por inicialización completa, hay que indicar los valores de todas las constantes. Por ejemplo:

```
enum Color {
  Rojo = "ROJO",   // "ROJO"
  Verde = "VERDE", // "VERDE"
  Azul = "AZUL"    // "AZUL"
}
```

Acceso a constantes de un *enum*

Para poder acceder a alguna de estas constantes, primero se debe acceder al enum y luego indicar la constante a acceder. Por ejemplo:

```
console.log(Color.Rojo) // "ROJO"
```

Para comparar alguna constante del enum frente a una variable, se puede usar directamente la constante, como se muestra en el siguiente ejemplo:

```
const miColorFavorito = Color.Rojo;

console.log(Color.Rojo === miColorFavorito ); // true siempre y cuando
no se modifique la constante miColorFavorito
```

Enums declarados como constantes

Es importante mencionar que también se pueden declarar enums como constantes, lo que fijará los valores que se le pongan en tiempo de compilación, agilizando el tiempo de ejecución. Por ejemplo:

```
const enum Color {
  Rojo = "ROJO",   // "ROJO"
  Verde = "VERDE", // "VERDE"
  Azul = "AZUL"    // "AZUL"
}
// Serán esos valores y no se podrán cambiar durante su ejecución
```

Unión de tipos

También, están los “enums” declarados como unión de tipos. Aquí vamos a ver también cómo declarar un *tipo*:

```
type Estados = "Prendido" | "Apagado";
```

Es similar a cuando tenemos una variable que puede ser de múltiples tipos de datos. No podremos hacer Estados.Prendido , pero podemos indicar que una variable es del tipo de datos Estados, que solo acepta los strings “Prendido” y “Apagado” como datos.

JSON

Qué es JSON

JSON es el acrónimo en inglés de *JavaScript Object Notation (Notación de objetos de JavaScript)*. Como sugiere el nombre, los JSON se utilizan comúnmente para representar objetos. La utilidad que se le da a los JSON es la de intercambiar datos, razón por la cual se lo considera un lenguaje de transporte. Esto implica que sirve para intercambiar información entre emisor y receptor. Es especialmente útil para objetos simples que solo tienen un conjunto limitado de propiedades.

Los objetos JSON se basan en el concepto de clave-valor, donde para poder acceder a un valor, se necesita la clave que lo define. Un ejemplo de esto puede ser *razaPerro: "Cocker Spaniel"*. La clave aquí vendría a ser *razaPerro*, y *"Cocker Spaniel"* es el valor del campo

Ejemplo de JSON

Un objeto JSON típico puede tener varios tipos de datos, como JSON anidados, arreglos, matrices, fechas, etc. A continuación se muestra un ejemplo de un objeto JSON que representa los datos de un perro en un criadero de perros de diferentes razas::

```
let perro = {
  "nombre": "Rufo",
  "raza": "Golden Retriever",
  "edad": 5,
  "comidasFavoritas": ["Solomillo", "Atún"],
  "gustaJugar": true,
  "vacunas": {
    "rabia": true,
    "moquillo_canino": true,
    "parvo": true
  }
}
```

Inferencia y limitaciones

Otra cuestión a considerar es que no le aclaramos a TypeScript el tipo de datos del perro. Esto es debido a que TypeScript puede inferirlos de manera implícita. Debido a esto, podemos notar que si queremos añadir un campo que antes no existía, aparecerá un error de compilación:

```
perro.edad = 27; // Ok
perro.hora = 5; // Error
```

Como podemos ver, nos permite cambiar la edad, pero al cambiar la hora, el compilador (o el IDE mismo en este caso) nos indica que la propiedad *hora* no existe en el tipo del objeto antes definido.

Interfaces

Qué son

De querer modelar un objeto JSON, arrays, una función o, próximamente, clases, se utilizan las interfaces. Estas son estructuras que definen el "contrato" de la aplicación y se introducen en TypeScript, lo que implica que no existen en JavaScript.

Creando nuestra primera interfaz

Estas se declaran con la palabra reservada *interface,* y admiten la definición de propiedades y métodos (funciones sobre el objeto). Para ello, hay que tener en cuenta un par de preguntas:

- ¿Cuáles son las claves o *keys* de cada campo?
- ¿Qué restricciones hay para cada campo?
- ¿Qué valores puede tomar cada campo?
- ¿Qué tipo de datos le corresponde a cada campo?
- El campo, ¿es opcional?

Por lo demás, el formato es similar al JSON.
Queremos modelar e implementar una estructura que nos permita representar a un perro en la vida real dentro de nuestro programa. Para ello, primero haremos un gráfico de lo que tendría el Perro:

Ahora, vamos a modelar la interfaz Perro en TypeScript:

```
interface IPerro { // Estamos definiendo al tipo de objeto Perro
  nombre: string;
  raza: string;
  edad: number;
  peso: number;
  color: string;
  comidasFavoritas: string[];
  gustaJugar: boolean;
  vacunas?: {
    rabia: boolean;
    moquilloCanino: boolean;
    parvo: boolean;
  }
}
```

Ampliando la interfaz Perro

De forma similar, podemos añadir la definición de la interfaz IVacunas y los datos de adopción de los perros padres. Establecemos el nombre como *readonly*, lo que implica que una vez establecido el nombre del perro, no se puede cambiar. Optamos por esta opción debido a que el nombre es identificatorio.

```
interface IVacunas{
    rabia: boolean;
    moquilloCanino: boolean;
    parvo: boolean;
}

interface IPerro {
  readonly nombre: string;
  raza: string;
  edad: number;
  comidasFavoritas: string[];
  gustaJugar: boolean;
  vacunas?: IVacunas;
  madre?: IPerro;
  padre?: IPerro;
}
```

Ahora que finalizamos de modelar al perro que hicimos, agregamos un par de cosas extra, para poder introducir el siguiente tema.

Agregado de métodos a la interfaz

Si pensamos en un perro dentro del criadero, podríamos también darle métodos que modifiquen o devuelvan su información. El perro puede adoptarse, se le puede asignar un nombre, agregarle información sobre sus padres, actualizar el historial clínico, etc. Entonces, ahora, veamos la transformación de la interfaz, ahora llamada IPerro, por Interfaz Perro:

```
interface IPerro {
  readonly nombre: string;
  raza: string;
  edad: number;
  comidas_favoritas: string[];
  vacunas?: IVacunas;
  madre?: IPerro;
  padre?: IPerro;
```

```
  crecer: () => void;
  sumarInformacionPadre: (padre: IPerro) => void;
  sumarInformacionMadre: (madre: IPerro) => void;
  sumarVacunas: (vacunas: IVacunas) => void;
}
```

Lo que añadimos son funciones. La función está definida sin usar *function*, y algunas reciben variables y todas devuelven void, o sea que no retornan ningún valor.

Clases

Introducción

Ahora que establecimos la estructura de nuestro objeto, podemos pasar a crear una clase. Una clase es una herramienta especial de algunos lenguajes de programación que funciona como un modelo o *template*. Puede *instanciarse* las veces que sea necesario, para crear diferentes objetos. Es importante mencionar que no es necesario tener una interfaz previa para crear una clase, aunque es una buena práctica y permite tener separado el modelo de su implementación. Además, nos permitiría tener múltiples interpretaciones de una interfaz en el futuro.

La clase tomará el modelo y creará un constructor donde se establezcan los valores de las variables y se implementen los métodos definidos en el modelo. Para crear una clase, usamos la palabra reservada *class*, y para basarnos en la interfaz creada anteriormente, usamos la palabra reservada *implements*.

Seguridad en datos

Hay dos tipos de seguridad en datos que vamos a ver: *public* y *private*. Por lo general, uno no debería tener permitido el acceso directo a un atributo, y mucho menos poder modificarlo. Para poder indicar que un atributo tiene acceso restringido, lo declaramos como *private*. Por otro lado, para que algunos atributos puedan ser accedidos sin problemas los declaramos como *public.*

Para obtener el valor de los atributos privados (supongamos el atributo privado X), implementamos un método *getX()*, invisible para la interfaz. Si este atributo se puede sobreescribir, entonces también se puede establecer un método *setX()*, también invisible para la interfaz.

Modelado de un Perro

Poniendo en práctica lo discutido hasta el momento, una clase válida para modelar un perro podría ser la siguiente:

```
class Perro implements IPerro {
    private _nombre: string;
    private _raza: string;
    private _edad: number;
    private _comidasFavoritas: string[];
    private _vacunas?: IVacunas;
```

```
    private _madre?: IPerro;
    private _padre?: IPerro;

    constructor(
        nombre: string,
        raza: string,
        edad: number,
        comidasFavoritas: string[],
        vacunas?: IVacunas,
        madre?: IPerro,
        padre?: IPerro,
    ) {
        this._nombre = nombre;
        this._raza = raza;
        this._edad = edad;
        this._comidasFavoritas = comidasFavoritas;
        this._vacunas = vacunas;
        this._madre = madre;
        this._padre = padre;
    }

    get nombre() {
        return this._nombre;
    }

    get raza() {
        return this._raza;
    }

    get edad() {
        return this._edad;
    }

    get comidas_favoritas() {
        return this._comidasFavoritas;
    }

    get vacunas() {
        return this._vacunas;
    }

    get madre() {
        return this._madre;
    }

    get padre() {
```

```
        return this._padre;
    }

    crecer() {
        this._edad++;
        this._peso += 2;
    }

    sumarInformacionPadre(padre: IPerro) {
        this._padre = padre;
    }

    sumarInformacionMadre(madre: IPerro) {
        this._madre = madre;
    }

    sumarVacunas(vacunas: IVacunas) {
        this._vacunas = vacunas
    }
}
```

Una vez creada la clase *Perro*, se puede instanciar, con la palabra reservada *new,* de la siguiente manera:

```
const miPerro = new Perro("Fido", "Golden Retriever", 5, "Dorado",
["carne", "queso"], true, { rabia: true, moquillo_canino: true, parvo:
true });
```

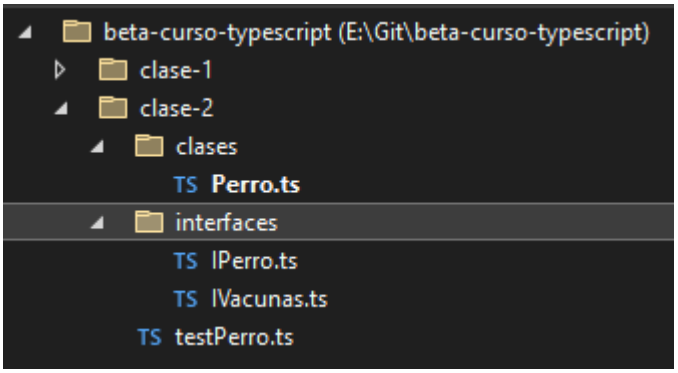
Ya creamos una instancia de Perro con el nombre "Fido", raza "Golden Retriever", 5 años de edad, comidas favoritas "carne" y "queso", tiene todas las vacunas, y sin padres.

Una vez que tenemos una instancia de Perro, podemos acceder a sus propiedades y métodos usando el operador de punto, por ejemplo:

```
console.log(miPerro.nombre); // imprime "Fido". No hay que escribir get
miPerro.crecer(); // hace que el perro crezca un año y aumente su peso
console.log(miPerro.edad); // imprime 6
```

Organización y separación de archivos

Una última sección del modelado de las clases y las interfaces es que todo debería estar separado en archivos distintos, para otorgar mayor organización y claridad. Cada interfaz debería separarse, y también la clase. Haremos una carpeta para las interfaces, una para clases (por más que solo tenemos una):



Dentro de cada archivo que nos queda, deberemos importar (Para importar, debemos tener un proyecto de nodejs creado, con npm init) las interfaces que utilicen, y exportar la interfaz o clase creada. Un ejemplo es dentro de la interfaz IPerro, que utiliza la interfaz IVacunas:

```
import IVacunas from "./Vacunas";

interface IPerro {
      // ...
 }

export default IPerro;
```