Introducción

En esta sección introduciremos el lenguaje de programación Typescript, la sintaxis que añade, el tipado, genéricos, decoradores, etc. Repasaremos además, algunos conceptos vistos previamente en el módulo de JavaScript a fin de comprender mejor la utilización del mismo y finalmente, profundizaremos el paradigma de programación orientada a objetos (POO) siempre orientandonos a la utilización del lenguaje para su aplicación en Angular.

**Objetivos**

* Comprender la diferencia entre Javascript y Typescript.
* Reconocer y escribir código typescript reconociendo los tipos de datos y las inferencias que realiza el compilador TSC.
* Comprender la importancia de la programación orientada a objetos y su diferencia con la programación estructurada. Comprender los fundamentos de la programación orientada a objetos.
* Crear código Typescript en base a los principios de la programación orientada a objetos para resolver problemas.

**TypeScript**

Typescript es un lenguaje de programación de código abierto creado por el equipo de Microsoft como una solución al desarrollo de aplicaciones de gran escala con Javascript dado que este último carece de clases abstractas, interfaces, genéricos, etc. y demás herramientas que permiten los lenguajes de programación tipados. Son ejemplos la compatibilidad con el  intellisense, la comprobación de tiempo de compilación, entre otras.

A typescripts se lo conoce además como un superset (superconjunto) de JavaScript ya que es un lenguaje que transpila el fuente de un lenguaje a otro  pero, incluye la ventaja de que es verdaderamente orientado a objetos y ofrece además, muchas de las cosas con las que estamos habituados a trabajar los desarrolladores como por ejemplo: interfaces, genéricos, clases abstractas, modificadores, sobrecarga de funciones, decoradores, entre otras varias.

 “*Los superset compilan en el lenguaje estándar, por lo que el desarrollador programa en aquel lenguaje expandido, pero luego su código es "transpilado" para transformarlo en el lenguaje estándar, capaz de ser entendido en todas las plataformas*”([desarrolloweb.com](https://desarrolloweb.com/articulos/introduccion-a-typescript.html))

Entonces, si posees algo de conocimiento de Javascript, ¡tienes ventaja!. Podemos cambiar los archivos de JavaScript a TypeScript de a poco e ir preparando la base del conocimiento para incorporar en su totalidad este nuevo lenguaje.

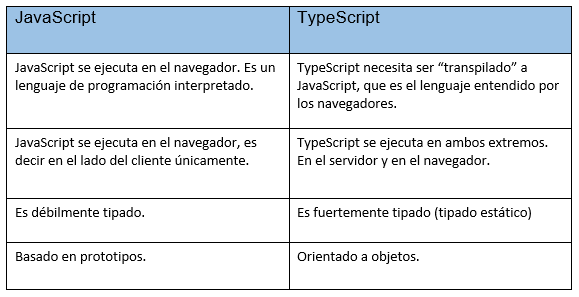


Tabla comparativa de los lenguajes JavaScript y TypeScript

## Tipado estático

Una de las principales características de typescript es que es fuertemente tipado por lo que, no sólo permite identificar el tipo de datos de una variable mediante una sugerencia de tipo sino que además permite validar el código mediante la comprobación de tipos estáticos.  Por lo tanto, TypeScript permite detectar problemas de código previo a la ejecución cosa que con Javascript no es posible.

Sugerencias para la escritura: Los tipos también potencian las ventajas de inteligencia y productividad de las herramientas de desarrollo, como IntelliSense, la navegación basada en símbolos, la opción Ir a definición, la búsqueda de todas las referencias, la finalización de instrucciones y la refactorización del código, podes consultar la documentación oficial [acá](https://docs.microsoft.com/es-es/learn/modules/typescript-get-started/2-typescript-overview).

Se agrega además que Typescript permite describir mucho mejor el fuente ya que, además de ser tipado,  es verdaderamente orientado a objetos (avanzaremos en esto más adelante) lo que permite a los desarrolladores un código más legible y mantenible.

 Variables

 Antes de avanzar en tipos y subtipos de datos en Typescript,  recordemos el concepto de variable.

Una **variable**es un espacio de memoria que se utiliza para almacenar un valor durante un tiempo (scope) en la ejecución del programa. La misma tiene asociado un tipo de datos y un identificador.

Debido a que Typescript es un superset de Javascript, la declaración de las variables se realiza de la misma manera que en Javascript:

***var:*** Es el tipo de declaración más común utilizada.

En este punto es importante agregar que si bien TypeScript es muy flexible y puede determinar el tipo de datos implícitamente, es aconsejable utilizar la nomenclatura que recomienda la página oficial (tipado estricto), ya que de este modo permitirá un mejor mantenimiento de nuestro código y el mismo será más legible.

Sintaxis:

C:\Users\54381\Downloads\image (15).png

***let***: Es un tipo de variable más nuevo (agregado por la [ECMAScript 2015](https://www.ecma-international.org/ecma-262/6.0/)). El mismo reduce algunos problemas que presentaba la sentencia var en las versiones anteriores de Javascript.

Este cambio permite declarar variables con ámbito de nivel de bloque y evita que se declare la misma variable varias veces, puedes profundizar en la documentación oficial haciendo clic acá

Sintaxis:

C:\Users\54381\Downloads\image (16).png

##### *****const:***** Es un tipo constante ya que, al asumir un valor no puede modificarse (agregado también por la *[ECMAScript 2015](https://www.ecma-international.org/ecma-262/6.0/)*)

 Sintaxis:

C:\Users\54381\Downloads\image (17).png

**Nota**: Como recordatorio, la diferencia entre ellas es que las declaraciones **let**se pueden realizar sin inicialización, mientras que las declaraciones **const**siempre se inicializan con un valor. Y las declaraciones **const**, una vez asignadas, nunca se pueden volver a asignar. (Referencia).

#### Inferencia de tipo en TypeScript

Typescript permite asociar tipos con variables de manera explícita o implícita como veremos a continuación:

Sintaxis de la inferencia explícita: C:\Users\54381\Downloads\image (18).png

cad

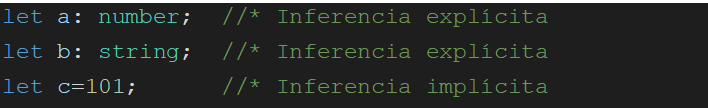
Ejemplo inferencia explícita: C:\Users\54381\Downloads\image (19).png

Ejemplo inferencia implícita: C:\Users\54381\Downloads\image (20).png

**Nota**: Si bien las asociaciones de tipo explícitas son opcionales en TypeScript, se recomiendan dado que permiten una mejor lectura y mantenimiento del código.

Veamos cómo funciona:

1. Abrir VSCode y crear un nuevo archivo titulado ***example01.ts***
2. En dicho archivo, escribir las siguientes declaraciones de variables:



En este caso, typescript interpreta que la variable **a** es del tipo number y **b** del tipo string dado que la declaración es explícita. En el caso de la variable **c** infiere que es del tipo number dado que éste es el tipo de datos que corresponde al valor con el que se ha inicializado la variable.

**Nota**: Observa que al posicionar el puntero del mouse sobre la variable c, VSCode abre un tooltip con la declaración explícita “**let c:number**”

 Pero ¿qué ocurre si intentamos asignar un tipo de datos diferente a la variable c?. Para ello, escribir a continuación la siguiente línea:

C:\Users\54381\Downloads\image (22).png

VSCode marca un error en la línea de la asignación y en el explorador puedes ver el archivo en rojo. Observa además que al posicionar el puntero del mouse sobre la línea VSCode  muestra el mensaje de error **“Type string is not assignable to type number”**

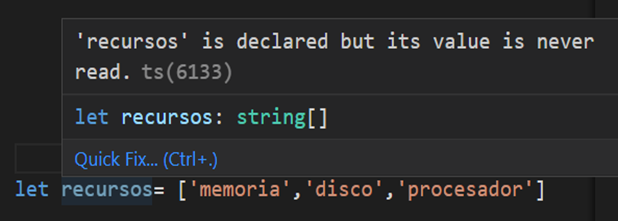


Error typescript. Asignación de tipos.

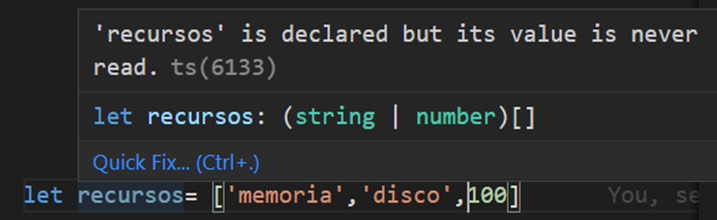
Analicemos otro ejemplo, dada la siguiente declaración:

C:\Users\54381\Downloads\image (24).png

Hasta aquí sabemos que: “let”, es la declaración de un arreglo cuyo nombre es “recursos” y el tipo no ha sido definido explícitamente.



Entonces, si  posicionamos el puntero del mouse sobre la variable “recursos” podemos observar que Typescript automáticamente entiende por sí solo que se trata de un arreglo del tipo String.  
  
Sin embargo, si luego introducimos en el array un valor de otro tipo, y posicionamos el puntero del mouse sobre la variable recursos, podremos observar que el arreglo admite variables del tipo “string” o (símbolo para “o” es “|”) “number”.

Y quizás, este no sea el comportamiento que deseamos por ello, se aconseja como buena práctica especificar el tipo de datos de manera explícita.

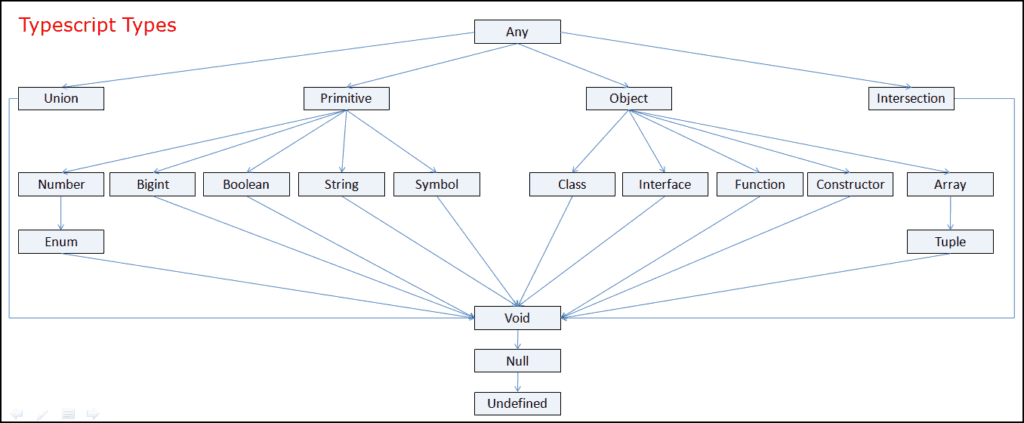
En este caso:

C:\Users\54381\Downloads\image (27).png

De esta manera, si intentamos introducir un elemento number u otro tipo a nuestro arreglo, el compilador nos marcará un error.

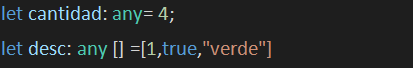
## Tipos de datos y subtipos

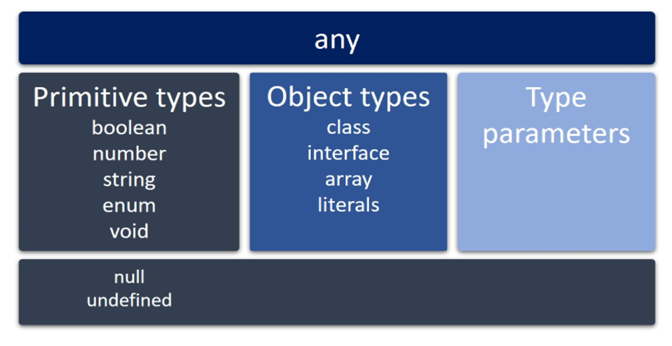
Todos los tipos en TypeScript son subtipos de un único tipo principal denominado tipo any. Any es un tipo que representa cualquier valor de JavaScript sin restricciones.



***Any:*** Puede ser de cualquier tipo y su uso está justificado cuando no tenemos información a priori de qué tipo de dato se trata. Este tipo de definición es propia de TypeScript.

Sintaxis:



Todos los demás tipos se clasifican como primitivos, de objeto o parámetros. Estos tipos presentan diversas restricciones estáticas en sus valores. 

#### Tipos de datos primitivos

Los tipos primitivos son: boolean, number, string, void, null, undefined y enum.

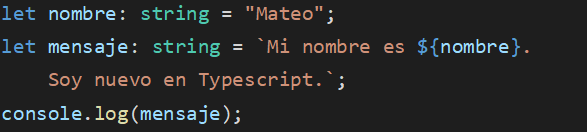
##### *****string:***** Representa valores de cadena de caracteres (letras);

Sintaxis:

C:\Users\54381\Downloads\image (30).png

TypeScript permite también usar plantillas de cadenas con las que podemos intercalar texto con otras variables:  ${ expr }

Ejemplo:



***number:*** Representa valores numéricos, como enteros (int) o decimales (float).

 Sintaxis:

C:\Users\54381\Downloads\image (32) (1).png

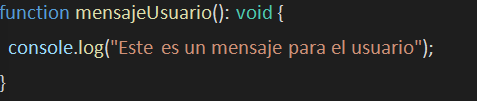
***boolean:*** Es un tipo de variable que puede tener solo dos valores, Verdadero (true) o Falso (false).

 Sintaxis:

##### C:\Users\54381\Downloads\image (33).png

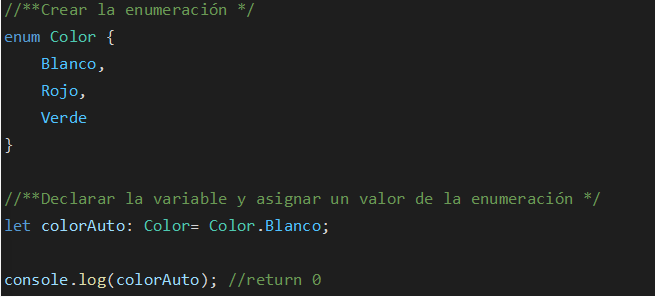
##### *Void:*El tipo void existe únicamente para indicar la ausencia de un valor, como por ejemplo en una función que no devuelve ningún valor.

Sintaxis:

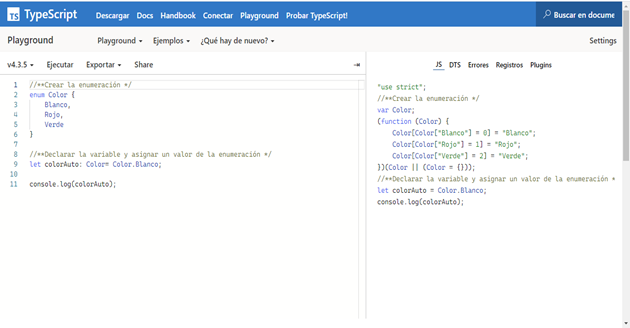


***Enum****:*Las enumeraciones ofrecen una manera sencilla de trabajar con conjuntos de constantes relacionadas. Un elemento enum es un nombre simbólico para un conjunto de valores. Las enumeraciones se tratan como tipos de datos y se pueden usar a fin de crear conjuntos de constantes para su uso con variables y propiedades.

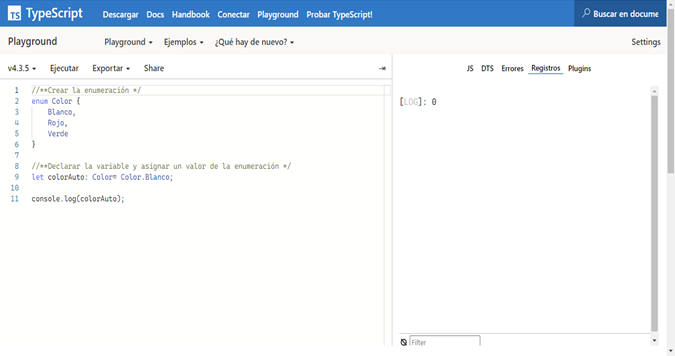
Siempre que un procedimiento acepte un conjunto limitado de variables, considere la posibilidad de usar una enumeración. Las enumeraciones hacen que el código sea más claro y legible, especialmente cuando se usan nombres significativos” ([referencia](https://docs.microsoft.com/es-es/learn/modules/typescript-declare-variable-types/4-enums))  
  
Ejemplo:



Para observar el fuente, lo que imprime la consola y el fuente javascript generado, hacer clic [aquí](https://www.typescriptlang.org/es/play?ssl=11&ssc=24&pln=1&pc=1#code/PTBUoYQJwUwQygAgDZ0TAdgVwLYynAMYCWAzxoqMALABQmuiEA9ss0gN52I+IBCqDIWYAabrwBKzAFajxPAGr4AJjDoBfbnRDgAIjEKoCSVIgBuCYnABGyGIgCeiOAGdiAcwwJEWChbZIqihoDHgEJOSUNLR2AC6IwgEAglixzABcTKzsALxZAQB0AnBCzADcWrTCGC6sMAVs7gAUiewpaQCUZUA).



El sitio [typescriptlang.org/play](https://www.typescriptlang.org/play) nos provee una herramienta para ver la transpilación a javascript e incluso ejecutarla. Para ello, simplemente hacer clic en Ejecutar:



Como puedes observar en las anteriores imagenes, tras ejecutar nuestro enum en typescript y transpilarlo a javascript, este se transforma el tipo enum (typescript)  a una función  (javascript) y devuelve un nro.  que va desde 0 en adelante, de acuerdo a la opción elegida.

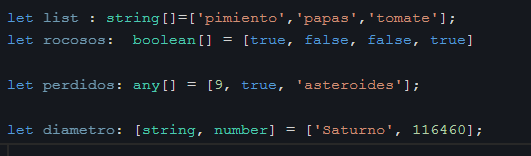
#### Tipos de objetos

Los tipos de objeto son todos los tipos de clase, de interfaz, de arreglos y literales.

**Nota**: Los tipos de clase e interfaz se abordarán más adelante en este mismo módulo.

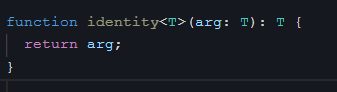
***Array:***Es un tipo de colección o grupos de datos (vectores, matrices).  El agrupamiento lleva como antecesor el tipo de datos que contendrá el arreglo.

Sintaxis:

. 

***Generic.****: También puedes definir tipos genéricos como sigue*

Sintaxis

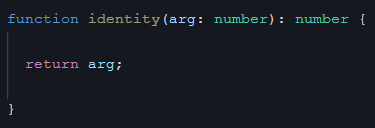


Los genéricos son como una especie de plantillas mediante los cuales podemos  aplicar un tipo de datos determinado a varios puntos de nuestro código. Sirven para aprovechar código, sin tener que duplicarlo por causa de cambios de tipo y evitando la necesidad de usar el tipo "any" ([referencia](https://desarrolloweb.com/articulos/generics-typescript.html)).

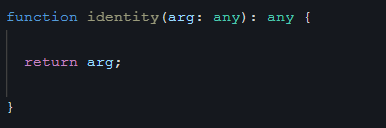
Los mismos se indican entre “mayores y menores” y pueden ser de cualquier tipo incluso clases e interfaces.

Veamos el ejemplo que nos provee la [fuente oficial](https://www.typescriptlang.org/docs/handbook/generics.html) de typescript:

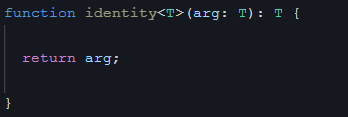
Si tenemos la siguiente función:



Pero, necesitamos que la misma sea válida para otros tipos de datos entonces podríamos cambiar el tipo number por any :



Sin embargo, el tipo any permite cualquier tipo de valor por lo que la función podría recibir un tipo number y devolver otro. Entonces, estamos perdiendo información sobre el tipo que debe devolver la función. Para solucionarlo, y obligar al compilador que respete el mismo tipo (parámetros de entrada y salida) podemos utilizar genéricos.



Observa que cambiamos any por la letra **T**.

T nos permite capturar el tipo de datos por lo que el tipo utilizado para el argumento es el mismo que el tipo de retorno.

##### *****Object*****: Es un tipo de dato que engloba a la mayoría de los tipos no primitivos.

C:\Users\54381\Downloads\image (10).png

##### Desestructuración****:****La desestructuración permite acceder a los valores de un array o un objeto.

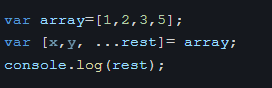
 Ejemplo - desestructuración de un objeto:

C:\Users\54381\Downloads\image (11).png

Ejemplo - desestructuración de un array:

C:\Users\54381\Downloads\image (12).png

Ejemplo - desestructuración con estructuración:

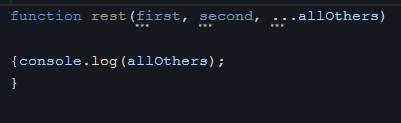


 Observa que la sintaxis **...rest**, nos permite agregar más parámetros. En este caso el resultado en consola será: [3, 5]

Puedes comprobar el fuente [aquí](https://www.typescriptlang.org/es/play?#code/G4QwTgBOYgngvAbQIwBoBMqDMqCsBdAbgFgAoUSRAD1VlQgDomwBTAZwBd94owZYSpAMYB7AHZsRAGxYMpIgOYAKVpwCUhIA) (clic en Ejecutar para observar lo que muestra en pantalla console.log)

**Estructuración:**  Como se pudo observar en el apartado anterior, la estructuración facilita que una variable del tipo array reciba una gran cantidad de parámetros.

Ejemplo en funciones:



Observa que la sintaxis **...allOthers** nos permite pasar más parámetros.

Luego, al llamar a la función con los siguientes parámetros:

C:\Users\54381\Downloads\image (15).png

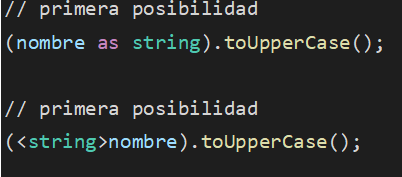
 Tipos null y undefined

“Los tipos null y undefined son subtipos de todos los demás tipos. No es posible hacer referencia explícita a los tipos null y undefined. Solo se puede hacer referencia a los valores de esos tipos mediante los literales null y undefined” ([referencia](https://docs.microsoft.com/es-es/learn/modules/typescript-declare-variable-types/2-types-overview)).

### **Aserción de tipos (As)**

Una aserción de tipos le indica al compilador "**confía en mí, sé lo que estoy haciendo**". Se parece al casting en otros lenguajes de programación pero no tiene impacto en tiempo de ejecución sino que le dice al compilador el tipo de datos en cuestión a fin de acceder a los métodos, propiedades, etc. del tipo de datos en tiempo de desarrollo.

 Sintaxis (dos posibles):



## Funciones

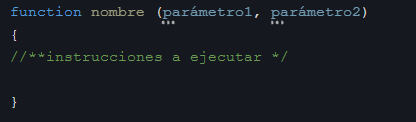
Una función es un **conjunto de instrucciones**o sentencias que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente y se caracterizan porque:

* ●      deben ser invocadas por su nombre.
* ●      permiten **simplificar el código**haciendo más legible y reutilizable.

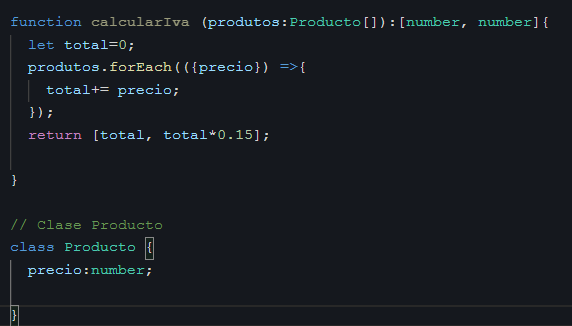
 La declaración de una función consiste en:

* ●       Un nombre
* ●       Una lista de parámetros o argumentos encerrados entre paréntesis.
* ●       Conjunto de sentencias o instrucciones encerradas entre llaves.

 Sintaxis:



 Ejemplo:



**Nota**: Puedes crear tus propias funciones y usarlas cuando sea necesario.

## TypeScript Programación Orientada a Objetos

A continuación abordaremos el tema de programación orientada a objetos ya que son de suma importancia para trabajar con Typescript y consecuentemente en Angular.

La programación orientada a objetos  (abreviada de ahora en más como POO), es un conjunto de reglas y principios de programación (o sea, un paradigma de programación) que busca representar las entidades u objetos del dominio (o enunciado) del problema dentro de un programa, de la forma más natural posible.

En el paradigma de programación tradicional o estructurado, que es el que hemos usado hasta aquí, el programador busca identificar los procesos (en forma de subproblemas o módulos) a fin de obtener los resultados deseados. Y esta forma de proceder no es en absoluto incorrecta: la estrategia de descomponer un problema en subproblemas es una técnica elemental de resolución de problemas que los programadores orientados a objetos siguen usando dentro de este nuevo paradigma. Entonces, ¿a qué viene el paradigma de la POO?

La programación estructurada se basa en descomponer procesos en subprocesos y programando cada uno como**rutina, función, o procedimiento** (todas formas de referirse al mismo concepto)  Sin embargo, esta forma de trabajar resulta difícil al momento de desarrollar sistemas realmente grandes o de mucha complejidad. Es decir que, la sola orientación a la descomposición en subproblemas no alcanza cuando el sistema es complejo dado que se vuelve difícil de visualizar su estructura general, se hace complicado realizar hasta las más pequeñas modificaciones sin que estas reboten en la lógica de un número elevado de otras rutinas, y por último es casi imposible replantear el sistema para agregar nuevas funcionalidades que permitan que ese sistema simplemente siga siendo útil frente a continuas nuevas demandas.

La POO significa una nueva visión en la forma de programar, buscando aportar claridad y naturalidad en la manera en que se plantea un problema. Ahora, el objetivo primario no es identificar procesos sino identificar actores: las entidades u objetos que aparecen en el escenario o dominio del problema, tales que esos objetos tienen no sólo datos asociados sino también algún comportamiento que son capaces de ejecutar. Por ejemplo, pensemos en un objeto como en un robot virtual: el programa tendrá muchos robots virtuales (objetos de software) que serán capaces de realizar eficiente y prolijamente ciertas tareas en las que serán expertos. Además, serán capaces de interactuar con otros robots virtuales (objetos...) con el objeto de resolver el problema que el programador esté planteando.

Ventajas de la POO

* ●      Es una forma más natural de modelar.
* ●      Permite manejar mejor la complejidad.
* ●      Facilita el mantenimiento y extensión de los sistemas.
* ●      Es más adecuado para la construcción de entornos GUI.

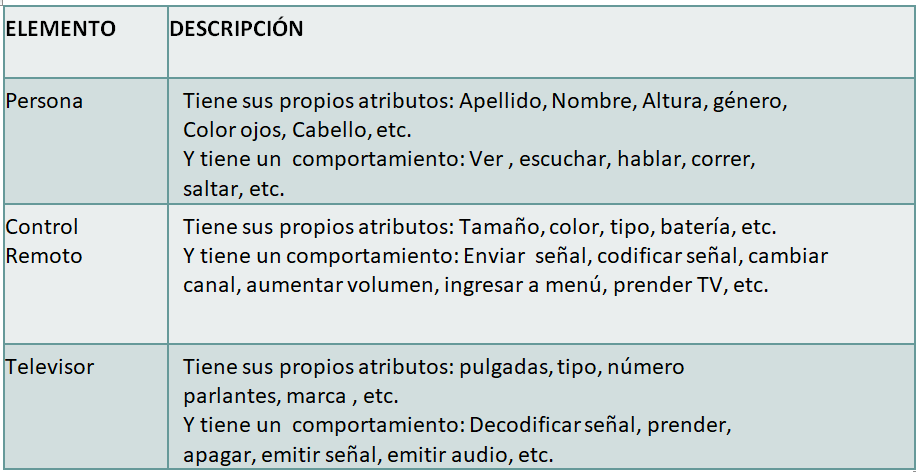
Fomenta la reutilización, con gran impacto sobre la productividad y confiabilidad.

## Objetos

 Para comprenderlo veamos la siguiente imagen y luego intentemos responder: ¿Cuáles son los objetos que se pueden abstraer para ver televisión? ¿Cómo podrías describirlos?



En el problema planteado se pueden identificar tres elementos: la persona, el televisor y el control remoto. Cada elemento  posee sus propias características y comportamientos. En la POO a estos  elementos se los conoce bajo el nombre de OBJETOS, a las características o estados que identifican a cada objeto ATRIBUTOS y, a los comportamientos o acciones, MÉTODOS.



Del ejemplo anterior podemos inferir el concepto de objeto:

**Un objeto es una entidad (tangible o intangible) que posee características propias (atributos) y acciones o comportamientos (métodos) que realiza por sí solo o interactuando con otros objetos.**

Sin embargo, además de las características (atributos) y acciones (métodos)… ¿Qué otras características podrías mencionar? Observa la siguiente imagen para analizar..



De acuerdo a la imagen anterior podemos decir que un objeto además, se  identifica por un nombre o identificador único que lo diferencia de los demás (en este caso puede ser el nro de serie), un estado (encendido, apagado), un tipo (televisor), nos abstrae dejándonos acceder sólo a las funciones encendido, apagado, cambiar canal, etc. y, por supuesto tiene un tiempo de vida.

En base a lo expuesto anteriormente podemos expresar las características generales de los objetos en POO:

     ●          Se identifican por un nombre o identificador único que lo diferencia de los demás.

     ●          Poseen estados.

     ●          Poseen un conjunto de métodos.

     ●          Poseen un conjunto de atributos.

     ●          Soportan el encapsulamiento (nos deja ver sólo lo necesario).

     ●          Tienen un tiempo de vida.

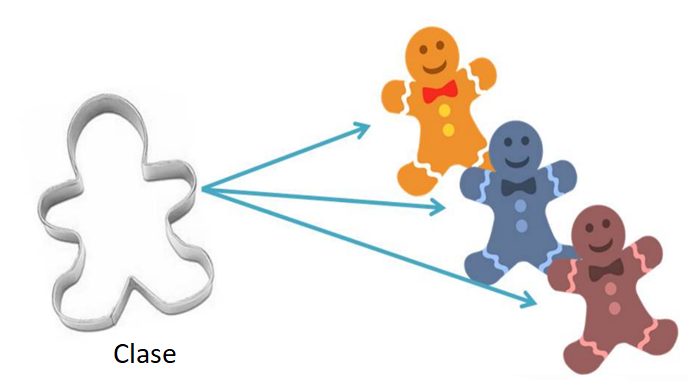
     ●          Son instancias de una clase (es de un tipo).

Para hacer eso, los lenguajes de programación orientados a objetos (como TypeScript) usan descriptores (plantillas) de entidades conocidas como *clases*.

Clases

Una *clase*es la descripción de una entidad u objeto de forma tal que pueda usarse como plantilla para crear muchos objetos que respondan a dicha descripción. Para establecer analogías, se puede pensar que una clase se corresponde con el concepto de *tipo de dato* de la programación estructurada tradicional, y los objetos creados a partir de la clase (llamados *instancias*en el mundo de la *POO*) se corresponden con el concepto de *variable* de la programación tradicional. Así como el *tipo*es uno solo y describe la forma que tienen todas las muchas *variables*de ese tipo, la *clase*es única y describe la forma y el comportamiento de los muchos *objetos*de esa clase.

Para describir objetos que responden a las mismas características de forma y comportamiento, se definen las *clases*.

Veamos el siguiente ejemplo:  


En el mismo podemos observar un molde para hacer galletitas (la clase) y el resultado que consiste en una o más galletas (objeto o instancia de clase).

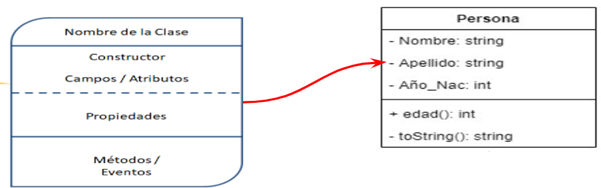
**Características generales de las clases en POO:**

* ●      Poseen un alto nivel de abstracción.
* ●      Se relacionan entre sí mediante jerarquías.
* ●      Los nombres de las clases deben estar en singular.

### **Representación gráfica de clases**

La representación gráfica de una o varias clases se realiza mediante los denominados Diagramas de Clase.  Para ello, se utiliza la notación que provee el Lenguaje de Modelación Unificado (UML, ver [www.omg.org](http://www.omg.org/) ), a saber:

* ●          Las clases se denotan como rectángulos divididos en tres partes. La primera contiene el nombre de la clase, la segunda contiene los atributos y la tercera los métodos.
* ●          Los modificadores de acceso a datos y operaciones, a saber: público, protegido y privado; se representan con los símbolos +, # y – respectivamente, al lado izquierdo del atributo. (+ público, # protegido, - privado).



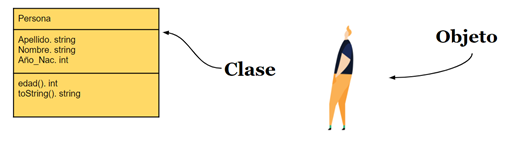
## Relación entre clases y objetos

Algorítmicamente, las clases son descripciones netamente estáticas o plantillas que describen objetos. Su rol es definir nuevos tipos conformados por atributos y operaciones. Es decir que,  las clases son una especie de molde de fábrica, en base al cual son construidos los objetos.

Por el contrario, los objetos son instancias particulares de una clase. Durante la ejecución de un programa sólo existen los objetos, no las clases.

* ●      La declaración de una variable de una clase NO crea el objeto.

La creación de un objeto, debe ser indicada explícitamente por el programador (instanciación),  de forma análoga a como inicializamos las variables con un valor dado, sólo que para los objetos se hace a través de un método CONSTRUCTOR.



## Clases en Typescript

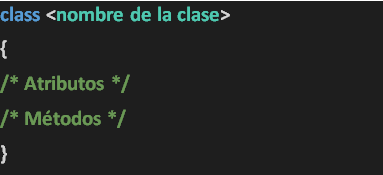
## Clases

Una clase en Typescript no es más que una secuencia de símbolos (o caracteres) de un alfabeto básico. Esta secuencia de símbolos forma lo que se denomina el código fuente de la clase. Hay dos aspectos que determinan si una secuencia de símbolos es correcta en Typescript: la **sintaxis**y la **semántica**.

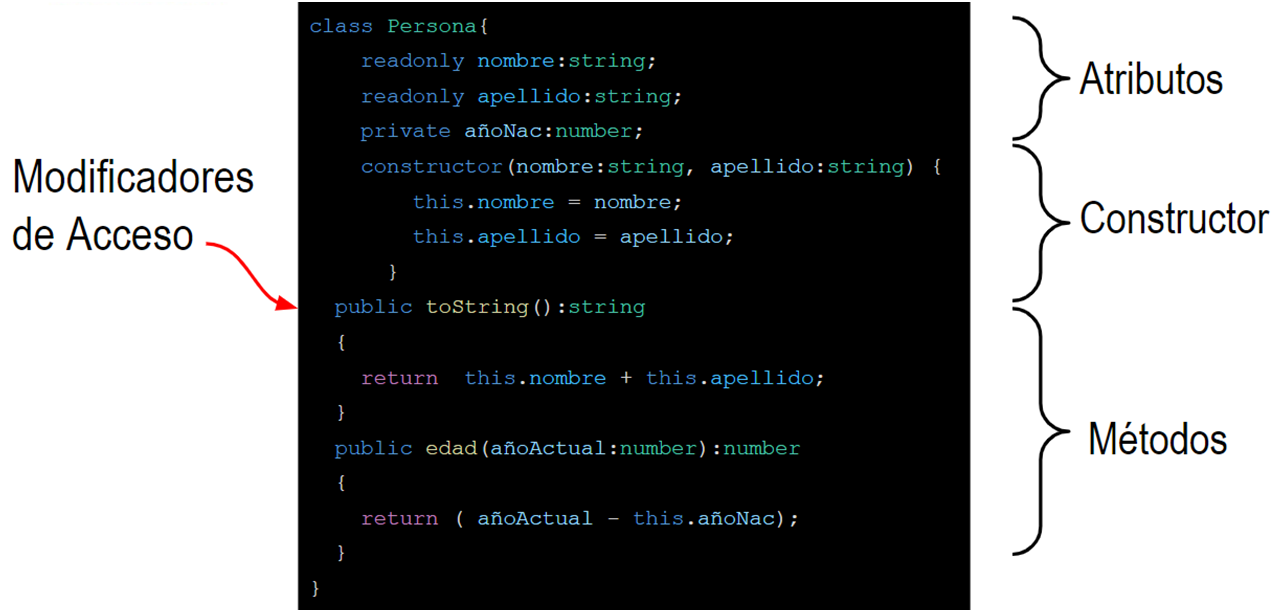
Las reglas de sintaxis de Typescript son las que permiten determinar de qué manera los símbolos del vocabulario pueden combinarse para escribir código fuente correcto mientras que la semántica por su parte,guarda una estrecha relación con las acciones o instrucciones lo que permite determinar el significado de la secuencia de símbolos para que se lleve a cabo la acción por la computadora.

Así, por ejemplo, como en el lenguaje natural son las reglas de la sintaxis las que nos permiten determinar que la siguiente secuencia “**robot el hombre programa**” no es correcta; las reglas semánticas nos posibilitan detectar errores de interpretación que no permiten que las acciones o instrucciones puedan ser ejecutadas. Ejemplo: “**el robot programa al hombre**” es sintácticamente correcta pero **no semánticamente correcta**.

Sintaxis para la definición de clases en Typescript:

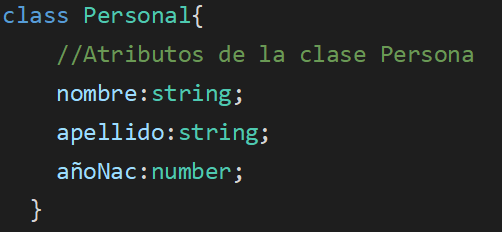


Dentro de las clases podemos encontrar:



**Atributos:** Son variables que se declaran dentro de la clase, y sirven para indicar la forma o características de cada objeto representado por esa clase. Los atributos, de alguna manera, muestran lo que cada objeto es, o también, lo que cada objeto tiene.

Sintaxis:  ***<nombre\_variable>: <tipo\_de\_datos>,*** ejemplo:



**Métodos:** Son funciones, procedimientos o rutinas declaradas dentro de la clase, usados para describir el comportamiento o las acciones de los objetos descriptos por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto hace.

La sintaxis es:

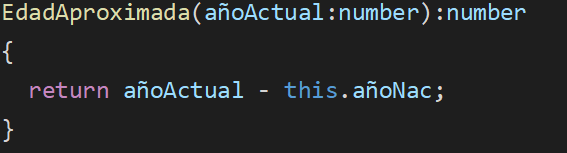
***<nombre\_método>(<parámetros>): <tipo\_de\_datos\_devuelto>,***

**{**

**//\*\*instrucciones\*/**

**}**

Ejemplo:



Dentro de estas <instrucciones> se puede acceder a todos los miembros definidos en la clase, a la cual pertenece el método. A su vez, todo método puede devolver un objeto como resultado de haber ejecutado las <instrucciones>. En tal caso, el tipo del objeto devuelto tiene que coincidir con el especificado en <TipoDevuelto>, y el método tiene que terminar con la cláusula return <objeto>;

Para el  caso que se desee definir un método que no devuelva objeto alguno se omite la cláusula return y se especifica void como <TipoDevuelto>.

Opcionalmente todo método puede recibir en cada llamada una lista de parámetros a los que podrá acceder en la ejecución de las <instrucciones> del mismo. En <Parámetros> se indican los tipos y nombres de estos parámetros y es mediante estos nombres con los que se deberá referirse a ellos en el cuerpo del método.

Es común (pero no obligatorio) que los atributos de la clase se declaren antes que los métodos. El conjunto de atributos y métodos de una clase se conoce como el conjunto de **miembros** de la clase.

Finalmente es importante mencionar que las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan relacionar entre sí, de manera que puedan compartir atributos y métodos sin necesidad de volver a escribirlos.

**Constructores:** es un método especial que permite instanciar un objeto. Su nombre está definido por la palabra **constructor**, y no tiene ningún tipo de retorno. Puede recibir 0 a n parámetros.

Sintaxis:

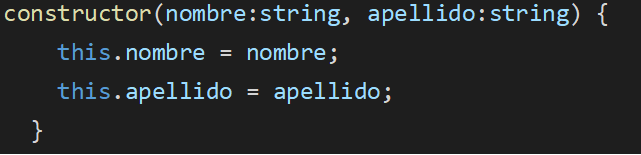
**Constructor(<parámetros>)**

**{**

**/\*\*instrucciones\*/**

**}**

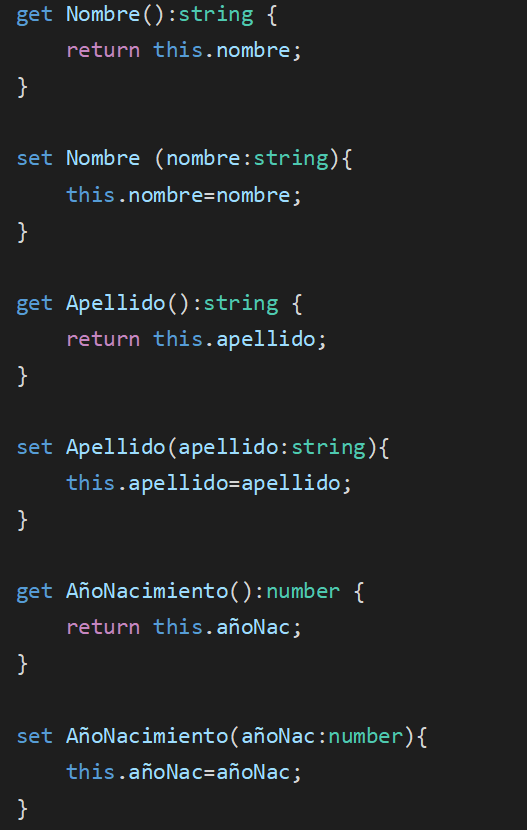
 Ejemplo:



Éste código suele usarse para la inicialización de los atributos del objeto a crear, sobre todo cuando el valor de éstos no es constante o incluye acciones más allá de una asignación de valor.

**Propiedades (getters y los setters).**Las mismas proporcionan la comodidad de los miembros de datos públicos sin los riesgos que provienen del acceso sin comprobar, sin controlar y sin proteger a los datos del objeto.

Ejemplo:

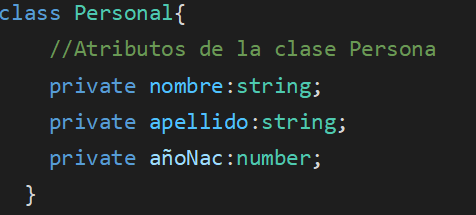


***Modificadores de acceso:*** La forma que los programas orientados a objetos, provee para que un programador obligue a respetar el Principio de Ocultamiento son los llamados *modificadores de acceso*.

Se trata de ciertas palabras reservadas que colocadas delante de la declaración de un atributo o de un método de una clase, hacen que ese atributo o ese método tengan accesibilidad más amplia o menos amplia desde algún método que no esté en la clase. Así, los modificadores de acceso pueden ser: *public*, *private* , *protected*

* ●      **Public** : un miembro público es accesible tanto desde el interior de la clase (por sus propios métodos), como desde el exterior de la misma (por métodos de otras clases).
* ●      **Private**: sólo es accesible desde el interior de la propia clase, usando sus propios métodos.
* ●      **Readonly**:  El acceso es de sólo lectura.
* ●      **Protected**: aplicable en contextos de herencia (tema que veremos más adelante), hace que un miembro sea público para sus clases derivadas y para clases en el mismo paquete, pero los hace privados para el resto.

 Sintaxis: ***<modificador> <atributo o método>,***ejemplo:



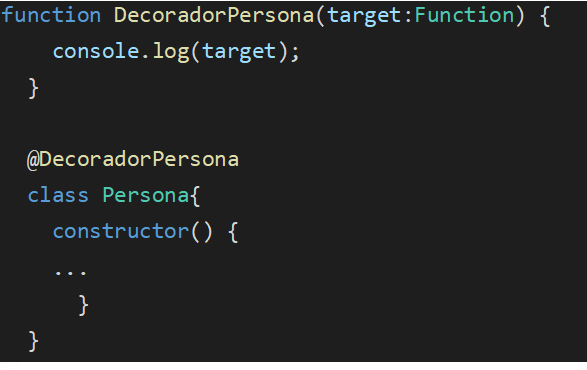
## Decoradores e instancias en TypeScript

### **Decoradores de Clase**

En Typescript, los decoradores (decorators en inglés) permiten  añadir anotaciones y metadatos o incluso cambiar el comportamiento de clases, propiedades, métodos y parámetros.

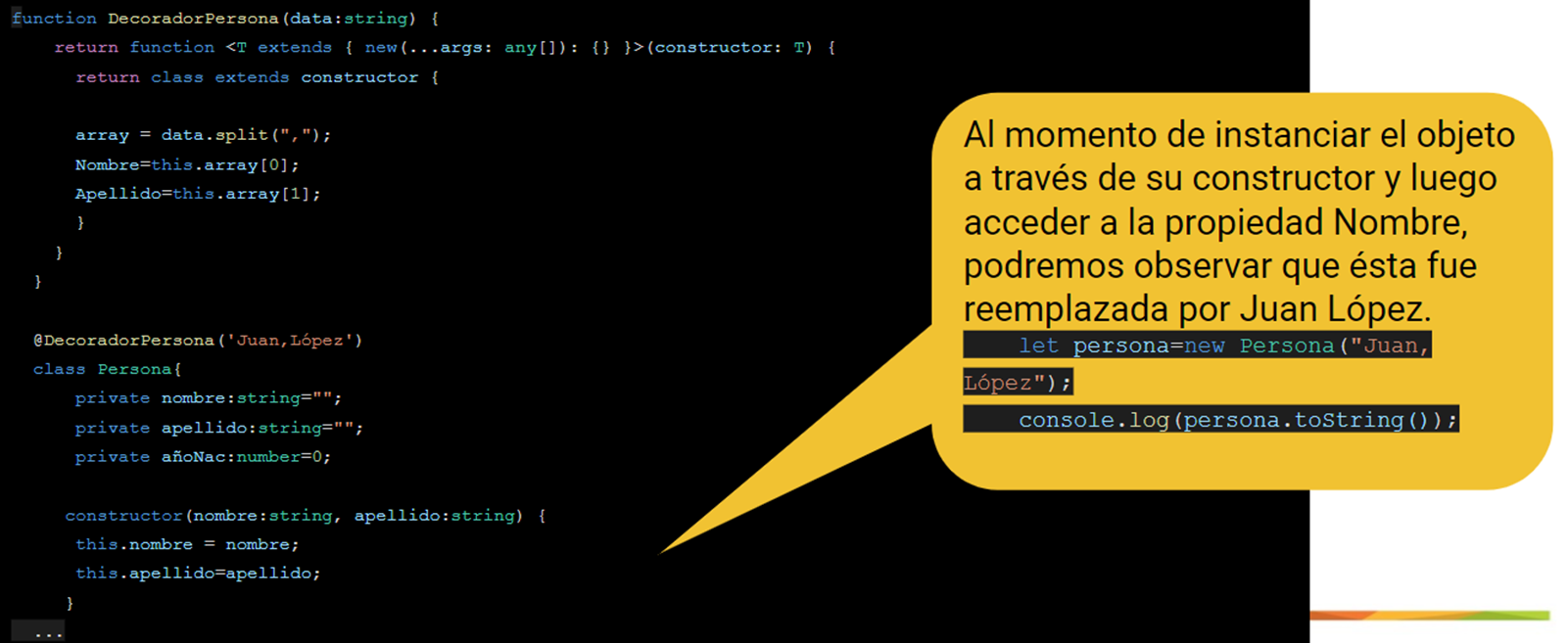
 Un decorador no es más que**una función** que, dependiendo de qué cosa queramos decorar, sus argumentos serán diferentes.

 Ejemplo:



En el ejemplo imprimimos por consola la clase Persona que fue decorada. Puedes ver y ejecutar el ejemplo [aqui](https://www.typescriptlang.org/es/play?" \l "code/GYVwdgxgLglg9mABAEQKYTgJwIYBMsAKqmAzgtgBRTaYDmqUAXAGLjTxgCUiA3gLAAoRMMQYwZADaoAdBLi0qNelE4BuQcIC+GxDoACaDDnyYipchogTsJEojNkw2fkJEAHTDABu2KKkRgcAC2AEaYqIwkUJ5gtAC8AEQJ6q7CHt6+-thuqBISMPiR0TCxick6aZ4+fojYAI9wAHLYEIxgIKHEcQAMKSIVYlGYINBYFIGh4UUxtAA0tTl5).

Si necesitamos algo más avanzado, debemos pasar parámetros a los decoradores:



Puedes ver  y ejecutar el fuente completo [aquí](https://www.typescriptlang.org/es/play?#code/GYVwdgxgLglg9mABAEQKYTgJwIYBMsAKqmAzgtgBS7ZTYBcJUmMYA5gJSIDeAsAFCJEmVFBCYkoSLASIAPABVEqAB5RUYXCW6IwqAO4UAdMeyZWJOomxgAngG0Auu0tcAvolcA+ChjCNMINBYlvKcvAKCQiJiSBAANtgkWipqGlq+-oFQWNz8kZF5kaY4NogAvIjUtIYkAA5xMFAUAEQANM3sANyFggBycAC2AEbCZVAAFjAkhsXY9gAMD).

**Nota**: Es posible también decorar métodos, propiedades, etc.

## Instancias

Para manipular los objetos o instancias de las clases (tipos) también se utilizan variables y éstas tienen una semántica propia la cual, se diferencia de los tipos básicos. Para ello, deberemos usar explícitamente el operador NEW. En caso contrario contendrán una referencia a null, lo que semánticamente significa que no está haciendo referencia a ningún objeto.

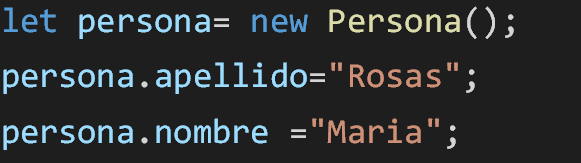
 Sintaxis para instanciar objetos: **<nombre\_objeto>= new <Nombre\_de\_Clase>(<parámetros>),**ejemplo:

C:\Users\54381\Downloads\image (3) (2).png

Sintaxis para inicializar un objeto:

 Hay 3 maneras de inicializar un objeto. Es decir, proporcionar datos a un objeto.

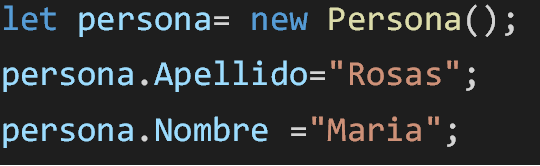
1. Por referencia a variables, ejemplo:

4

1. Por medio del constructor de la clase, ejemplo:

C:\Users\54381\Downloads\image (5) (3).png

1. Por medio de la propiedad setter, ejemplo:



## Recomendaciones

 Aunque cada programador puede definir su propio estilo de programación, una buena práctica es seguir el estilo utilizado por los diseñadores del lenguaje pues, de seguir esta práctica será mucho más fácil analizar el fuente de terceros y, a su vez, que otros programadores analicen y comprendan nuestro fuente.

* ●          Evitar en lo posible líneas de longitud superior a 80 caracteres.
* ●          Indentar los bloques de código.
* ●          Utilizar identificadores nemotécnicos, es decir, utilizar nombres simbólicos adecuados para los identificadores lo suficientemente autoexplicativos por sí mismos para dar una orientación de su uso o funcionalidad de manera tal que podamos hacer más claros y legibles nuestros códigos.
* ●          Los identificadores de clases, módulos, interfaces y enumeraciones deberán usar **PascalCase**.
* ●          Los identificadores de objetos, métodos, instancias, constantes y propiedades de los objetos deberán usar camelCase.
* ●          Utilizar comentarios, pero éstos seguirán un formato general de fácil portabilidad y que no incluya líneas completas de caracteres repetidos. Los que se coloquen dentro de bloques de código deben aparecer en una línea independiente indentada de igual forma que el bloque de código que describen.

## Fundamentos del enfoque orientado a objetos (EOO)

El Enfoque Orientado a Objeto se basa en cuatro principios que constituyen la base de todo desarrollo orientado a objetos. Estos principios son:

* ●      Jerarquías (herencia, agregación y composición)
* ●      Abstracción
* ●      Encapsulamiento
* ●      Modularidad

 Otros elementos para destacar (aunque no fundamentales) son:

* ●      Polimorfismo
* ●      Tipificación
* ●      Concurrencia
* ●      Persistencia.

## Jerarquías

Las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan relacionar entre sí de manera que puedan compartir atributos y métodos sin necesidad de volver a escribirlos y así resolver un problema.

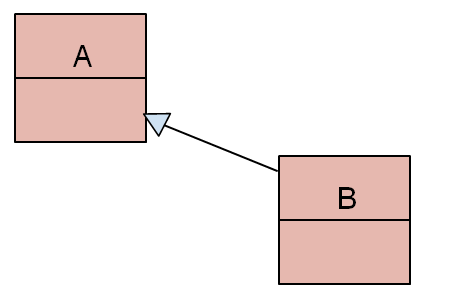
La posibilidad de establecer jerarquías entre las clases es una característica que diferencia esencialmente la programación orientada a objetos de la programación tradicional, ello debido fundamentalmente a que permite extender y reutilizar el código existente sin tener que volver a escribirlo cada vez que se necesite.

### **Herencia**

 En Programación Orientada a Objetos se llama herencia al mecanismo por el cual se puede definir una nueva clase B en términos de otra clase A ya definida, pero de forma que la clase B obtiene todos los miembros definidos en la clase A sin necesidad de hacer una redeclaración explícita. El sólo hecho de indicar que la clase B hereda (o deriva) desde la clase A, hace que la clase B incluya todos los miembros de A como propios (a los cuales podrá acceder en mayor o menor medida de acuerdo al calificador de acceso [public, private, protected, "default"] que esos miembros tengan en A).

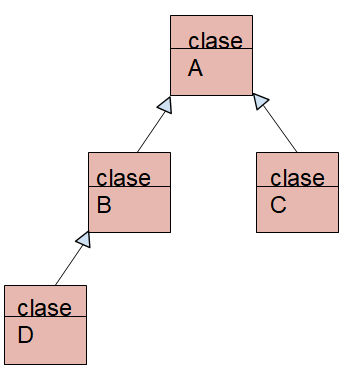
Es decir que la herencia permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos (Programación Diferencial), evitando repetición de código y permitiendo la reusabilidad.  


Cuando la clase B hereda de la clase A, se dice que hay una relación de herencia entre ellas, y se modela en UML con una flecha continua terminada en punta cerrada. La flecha parte de la nueva clase (o clase derivada) que sería B en nuestro ejemplo, y termina en la clase desde la cual se hereda (que es A en nuestro caso):



La clase desde la cual se hereda, se llama**super clase**, y las clases que heredan desde otra se llaman subclases o clases derivadas: de hecho, la herencia también se conoce como derivación de clases.

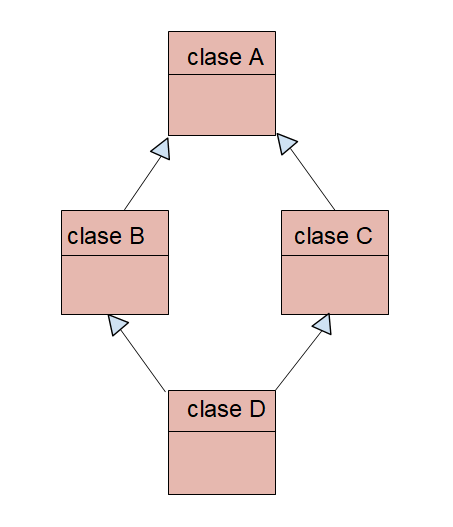
Una **jerarquía** de claseses un conjunto de clases relacionadas por herencia. La clase en la cual nace la jerarquía que se está analizando se designa en general como clase base de la jerarquía. La idea es que la clase base reúne en ella características que son comunes a todas las clases de la jerarquía, y que por lo tanto todas ellas deberían compartir sin necesidad de hacer redeclaraciones de esas características. El siguiente gráfico muestra una jerarquía de clases:



En esta jerarquía, la *clase base*es “*Clase A”*. Las clases “*Clase B”*y “*Clase C”*son *derivadas directas*de “*Clase A”*. Note que “*Clase D”*deriva en forma directa desde “*Clase B”*, pero en *forma indirecta*también deriva desde “*Clase A”*, por lo tanto todos los elementos definidos en “*Clase A”*también estarán contenidos en “*Clase D”*.  Siguiendo con el ejemplo, “*Clase B”*es super clase de “*Clase D”*, y “*Clase A”*es super clase de “*Clase B”*y “*Clase C”*.

 En general, se podrían definir esquemas de jerarquías de clases en base a dos categorías o formas de herencia:

***Herencia simple***: Si se siguen reglas de *herencia simple*, entonces *una clase puede tener una y sólo una superclase directa*. El gráfico anterior es un ejemplo de una jerarquía de clases que siguen *herencia simple*. La *Clase D*tiene una sola superclase directa que es *Clase B*. No hay problema en que a su vez esta última derive a su vez desde otra clase, como *Clase A*en este caso. El hecho es que en *herencia simple*, a nivel de gráfico UML, sólo puede existir *una*flecha que parta desde la clase derivada hacia alguna superclase.

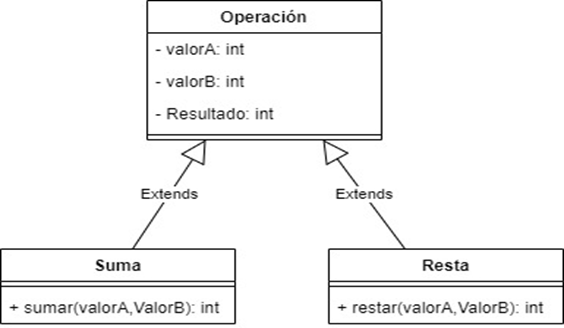
***Herencia múltiple***: Si se siguen reglas de *herencia múltiple*, entonces *una clase puede tener tantas superclases directas como se desee*. En la gráfica UML, puede haber varias flechas partiendo desde la clase derivada hacia sus superclases. El siguiente esquema muestra una jerarquía en la que hay *herencia múltiple*: note que *Clase D*deriva en forma directa desde las clases *Clase B*y *Clase C*, y esa situación es un caso de herencia múltiple. Sin embargo, note también que la relación que existe entre las *Clase B*y *Clase C*contra *Clase A*es de *herencia simple*; tanto *Clase B*como *Clase C*tienen una y sólo una superclase directa: *Clase A*.  


No todos los lenguajes orientados a objetos soportan (o permiten) la herencia múltiple: este mecanismo es difícil de controlar a nivel de lenguaje y en general se acepta que la herencia múltiple lleva a diseños más complejos que los que se podrían obtener usando sólo herencia simple y algunos recursos adicionales como la implementación de clases de interface.

Sabemos que una relación de uso implica que un objeto de una clase usa a un objeto de otra. Pero una relación de herencia implica que un objeto b de una clase B, es a su vez un objeto de otra clase A.

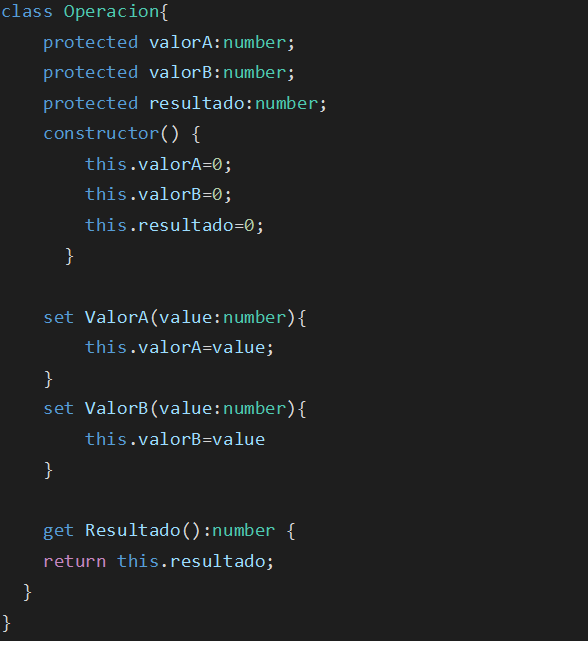
Veamos un ejemplo en Typescript:

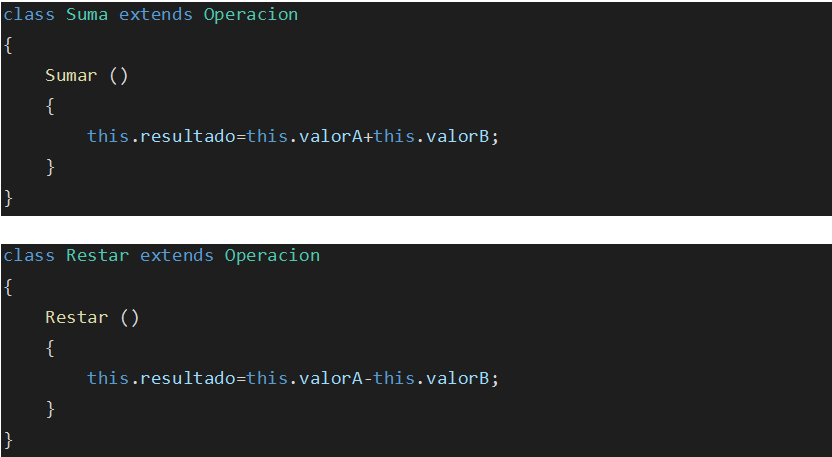
Necesitamos  crear dos clases que llamaremos Suma y Resta que derivan de una superclase llamada Operación:



En typescript, seguir los siguientes pasos:

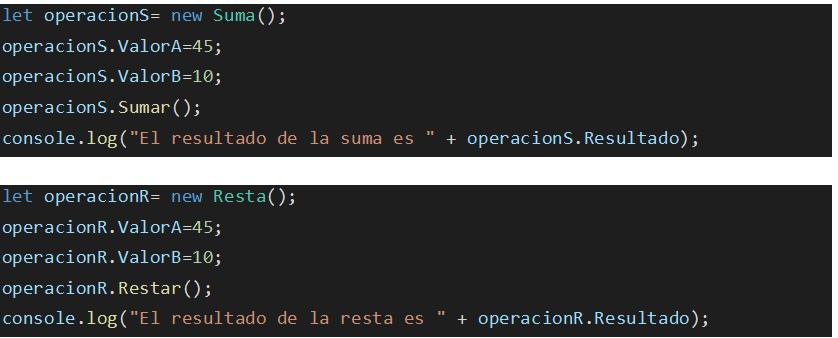
 1- Definir la superclase operación:



2- Luego, extender las subclases suma y resta: 

Observa que debemos utilizar la palabra “**extends**”

 3- Crear instancias de la clase suma y resta:



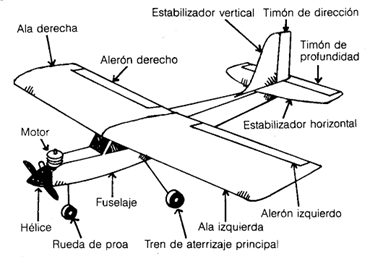
Puedes ver y evaluar el fuente [aquí](https://www.typescriptlang.org/es/play?#code/MYGwhgzhAEDyAOBTATmYBLA9gOwN4FgAoaE6eZTAF0WGoBNoA3METZAQQC5sBXAWwBGKANxFSZCtVqIGzVsgBC3fkOSjipclRr1oyRBB4hKYOpmWCRY0sBwRKyHrTYAKAJTRcJa+JKUAFugQAHRybOwAvAAM6r6kAUGhLGwK0bFx0Akh+obGpphpPqQAvmJFEIiU0ABqyRwucjyIFqpuBBpxWUnykY2I6SSlHdAVVbXyCg0sTS0obUXiXW).

**Nota**: observa que los modificadores de acceso en la superclase son **“protected”**. Estos permiten que la subclase pueda acceder a ellos y manipularlos.

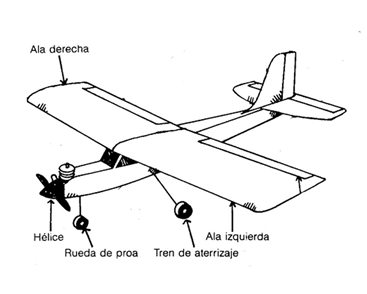
## Agregación y Composición

Las jerarquías de agregación y composición son asociaciones entre clases del tipo “es parte de”.

Para comprenderlo observemos la siguiente imagen de un aeroplano:



El mismo está compuesto de partes que pueden ser elementales (composición) o no (agregación):

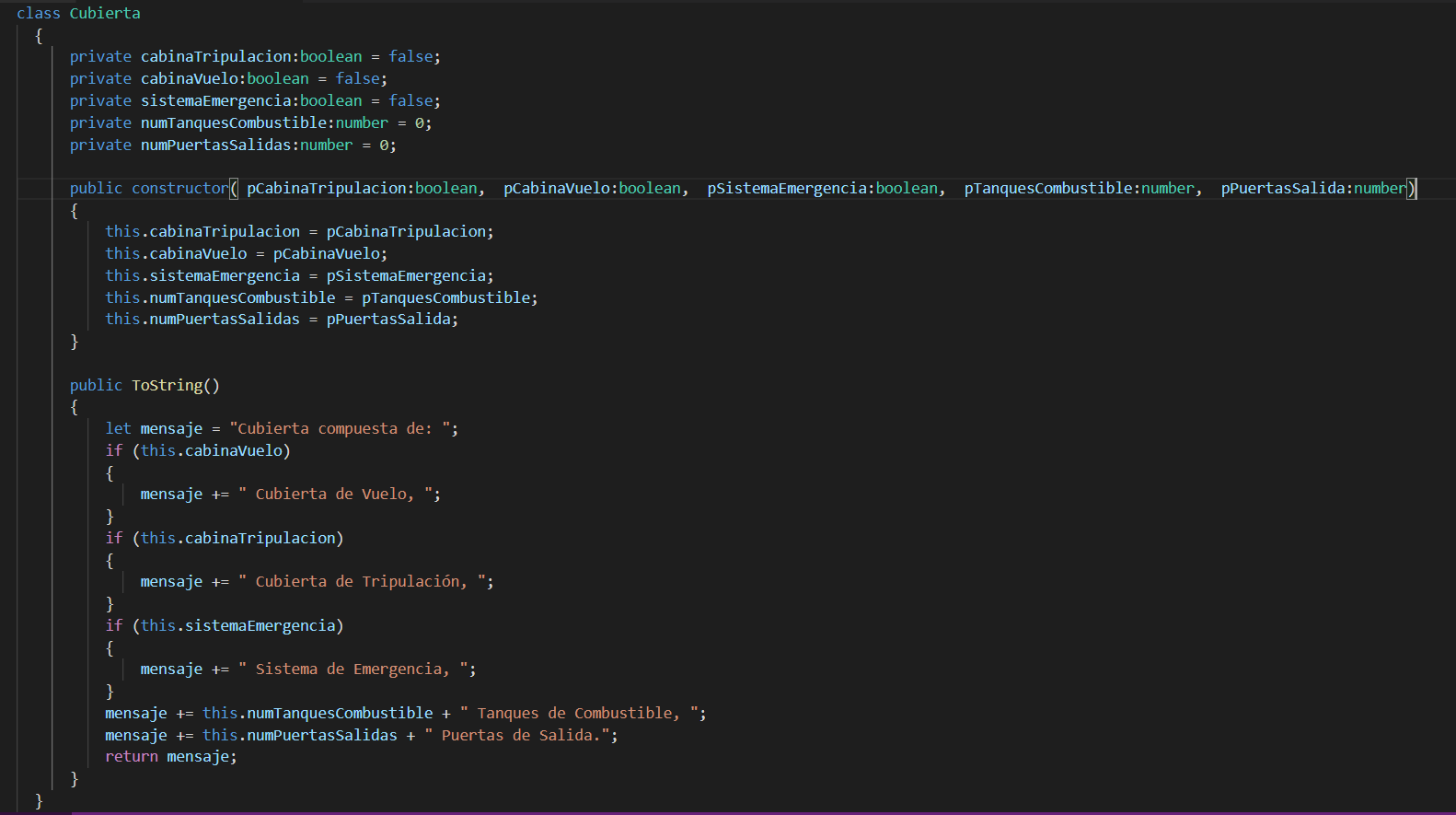
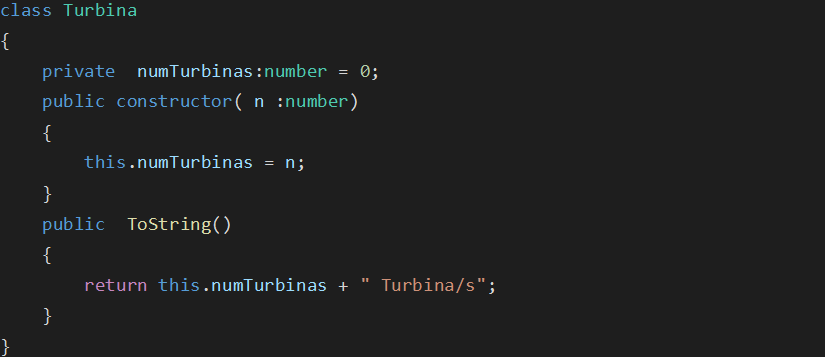


En la imagen anterior podemos observar que el aeroplano está compuesto por:

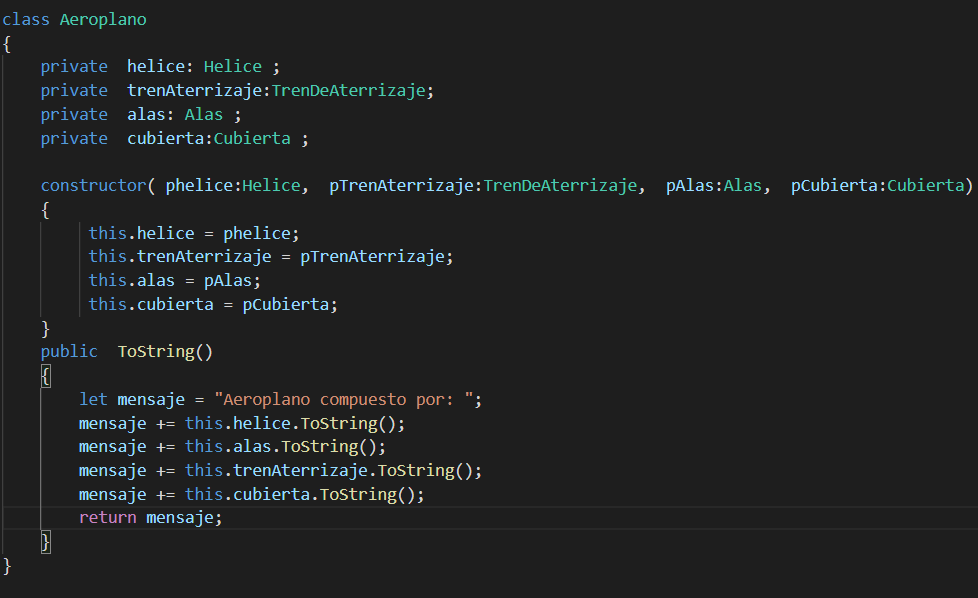
* ●      1 hélice frontal.
* ●      Tren de aterrizaje fijo, tiene 3 neumáticos y 3 amortiguadores.
* ●      2 alas frontales y 3 de cola.
* ●      La cubierta cuenta con sólo una cabina de vuelo, 1 tanque de combustible, 1 puerta de salida.

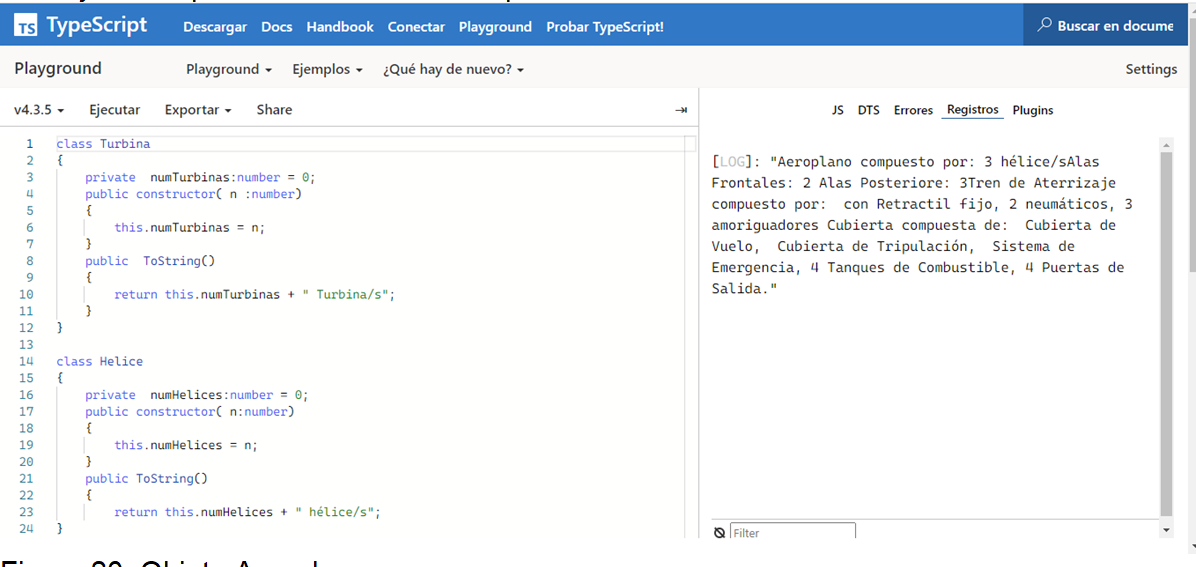
 Para resolverlo en Typescript, deberemos primero crear las clases Helice, TrenDeAterrizaje, Turbina, Cubierta, Alas, etc.

 A continuación ejemplo de las clases Turbina y Cubierta en Typescript



Las mismas forman parte elemental del Aeroplano (asociación de composición). Esta última se muestra a continuación:

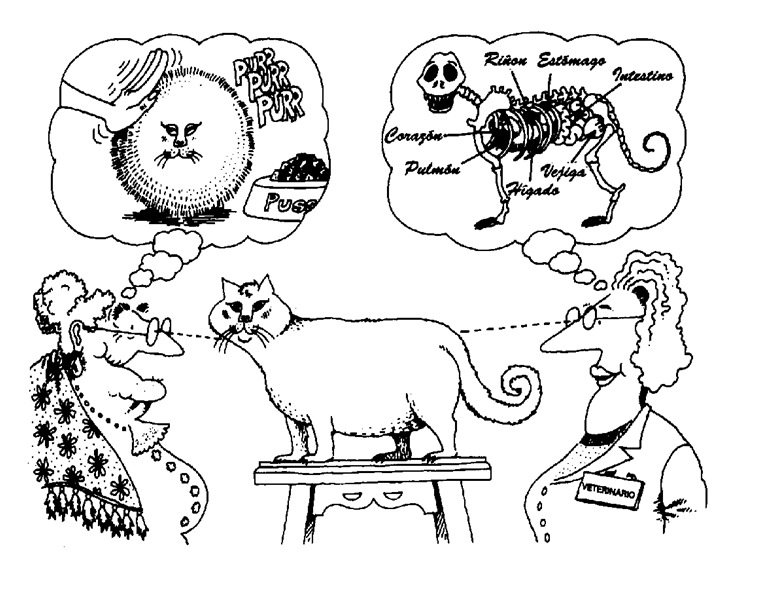
Si lo ejecutas el código podrás observar su composición:



Abstracción

Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo distingue de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

“*Una abstracción se centra en la visión externa de un objeto por lo tanto sirve para separar el comportamiento esencial de un objeto de su implementación. La decisión sobre el conjunto adecuado de abstracciones para determinado dominio es el problema central del diseño orientado a objetos. Se puede caracterizar el comportamiento de un objeto de acuerdo a los servicios que presta a otros objetos, así como las operaciones que puede realizar sobre otros objetos*”



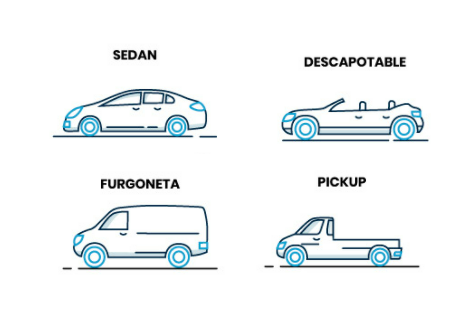
Los mecanismos de abstracción usados en el EOO para extraer y definir las abstracciones son:

* 1- La **GENERALIZACIÓN**. Mecanismo de abstracción mediante el cual un conjunto de clases de objetos son agrupados en una clase de nivel superior (Superclase), donde las semejanzas de las clases constituyentes (Subclases) son enfatizadas, y las diferencias entre ellas son ignoradas.

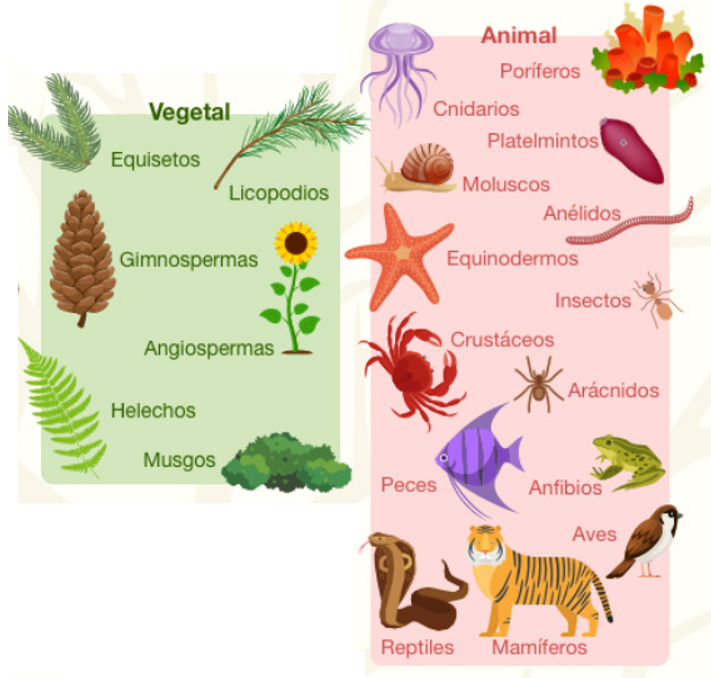
En consecuencia, a través de la generalización:

* + ●  La superclase almacena datos generales de las subclases
  + ●  Las subclases almacenan sólo datos particulares.
* 2- La **ESPECIALIZACIÓN**es lo contrario de la generalización. La clase Médico es una especialización de la clase Persona, y a su vez, la clase Pediatra es una especialización de la superclase Médico.
* 3- La **AGREGACIÓN**. Mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). Mediante agregación se puede definir por ejemplo un computador, por descomponerse en: la CPU, la ULA, la memoria y los dispositivos periféricos. El contrario de agregación es la descomposición.
* 4- La **CLASIFICACIÓN**. Consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La ejemplificación es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular

La clasificación es el medio por el que ordenamos, el conocimiento ubicado en las abstracciones, miremos algunos ejemplos en las siguientes imágenes.



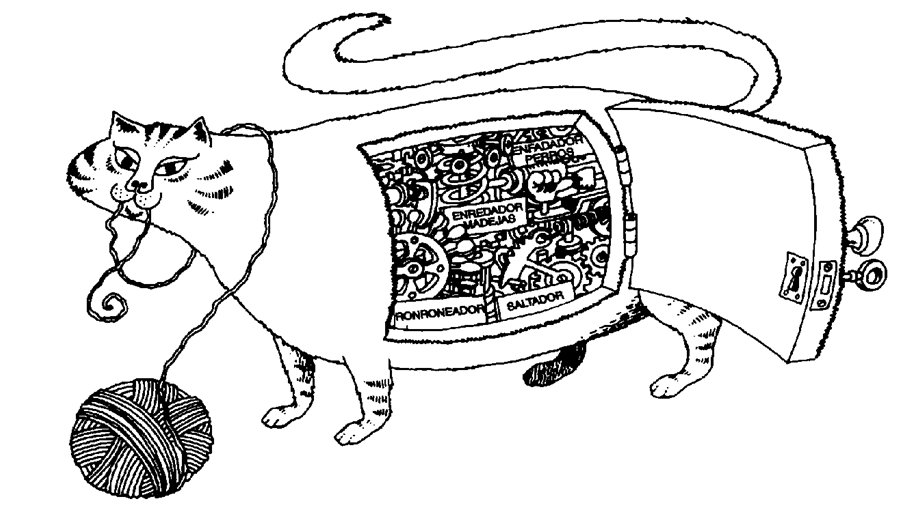
En resumen, las clases y objetos deberían estar al nivel de abstracción adecuado. Ni demasiado alto ni demasiado bajo



## Encapsulamiento

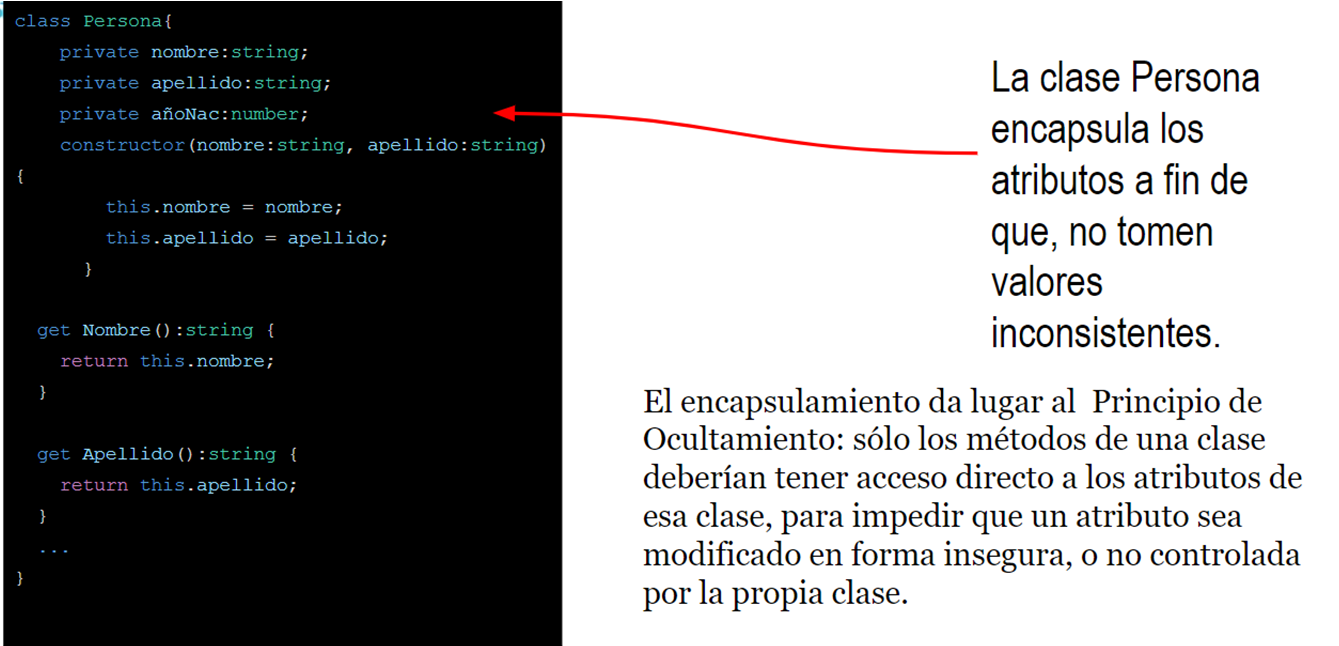
También conocido como, Ocultamiento. Es la propiedad de la POO que permite ocultar los detalles de implementación del objeto mostrando sólo lo relevante. Esta parte de código oculta pertenece a la parte privada de la clase y no puede ser accedida desde ningún otro lugar.

 El encapsulamiento da lugar al ya citado Principio de Ocultamiento: sólo los métodos de una clase deberían tener acceso directo a los atributos de esa clase, para impedir que un atributo sea modificado en forma insegura, o no controlada por la propia clase. El Principio de Ocultamiento es la causa por la cual en general los atributos se declaran como privados (private), y los métodos se definen públicos (public). Los calificadores private y public (así como protected , que se verá más adelante) tienen efecto a nivel de compilación: si un atributo de una clase es privado, y se intenta acceder a él desde un método de otra clase, se producirá en error de compilación.



Laura Álvarez, Helmer Avendaño, Yeison García, Sebastián Morales, Edwin Bohórquez, Santiago Hernandez, Sebastián Moreno, Cristian Orjuela (2017)

Ejemplo:



Modularidad

Es la propiedad que permite tener independencia entre las diferentes partes de un sistema. La modularidad consiste en dividir un programa en módulos o partes, que pueden ser compilados separadamente, pero que tienen conexiones con otros módulos. En un mismo módulo se suele colocar clases y objetos que guarden una estrecha relación. El sentido de modularidad está muy relacionado con el ocultamiento de información.



## Polimorfismo

Clases diferentes (polimórficas) implementan métodos con el mismo nombre. En resumen, el polimorfismo permite comportamientos diferentes, asociados a objetos distintos compartiendo el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento



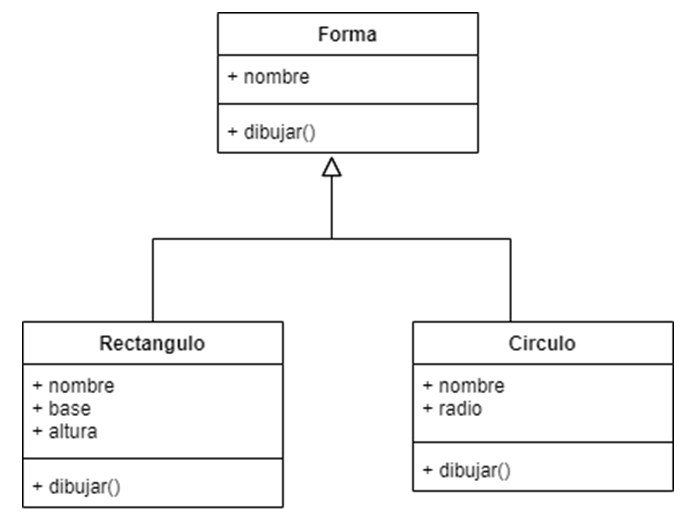
En el dibujo anterior puede verse la esencia del polimorfismo dado que todos implementan el método hablar aunque son diferentes objetos.

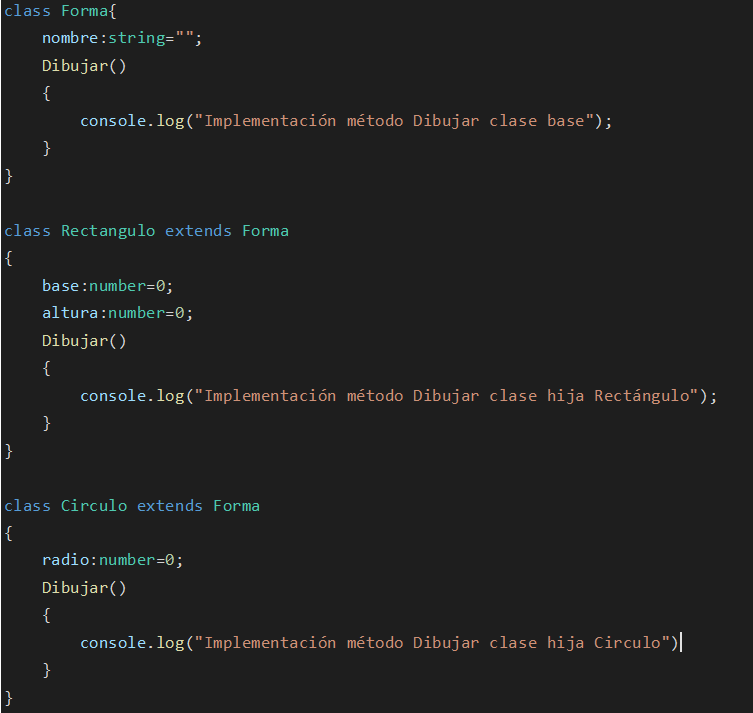
 Existen diferentes formas de implementar el polimorfismo en typescript:

### **Polimorfismo por herencia**

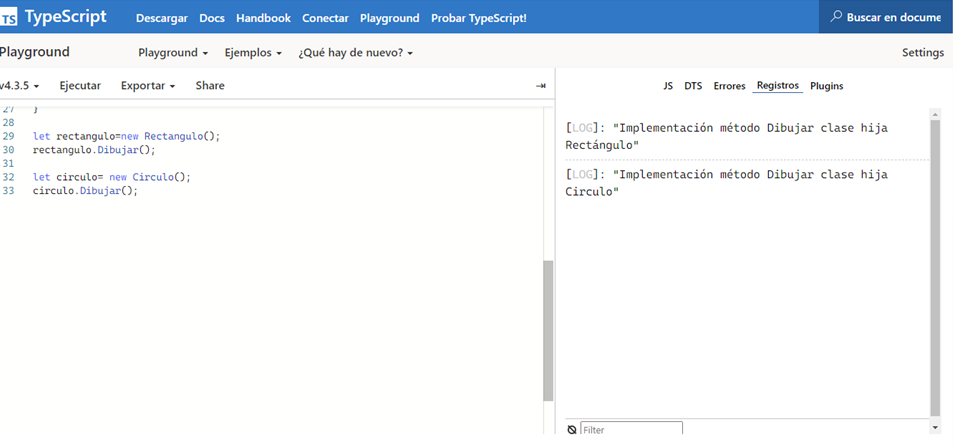
Cuando una subclase hereda de una clase base, obtiene todos los métodos, campos, propiedades y eventos de la superclase sin embargo, quizás necesitemos un comportamiento diferente para las clases derivadas (o subclases).

Ejemplo:

Del diagrama de clases anterior podemos deducir que no será lo mismo dibujar un rectángulo que un círculo por lo que el comportamiento deberá ser distinto (polimorfismo).

Ejemplo de polimorfismo en base a herencia en Typescript:  


Si creamos una instancia de la clase Rectangulo y Circulo y luego llamamos el método Dibujar observaremos que el mismo fue reemplazado por la implementación correspondiente para la forma:

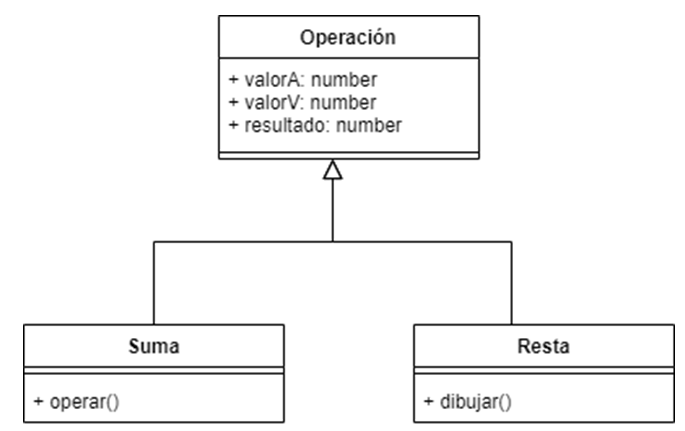


### **Polimorfismo por abstracción**

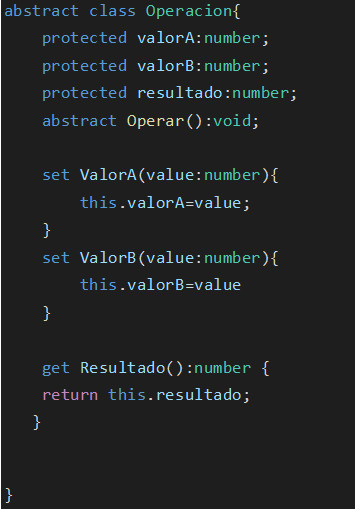
El polimorfismo por abstracción consiste en definir clases base abstractas (que no se pueden instanciar) pero que sirven de base para las clases derivadas. Es decir, sólo existen para ser heredadas.

**Nota**: Las clases abstractas no están implementadas o si lo están es parcialmente. Forzosamente se ha de derivar si se desea crear objetos de la misma ya que no es posible crear instancias  a partir de ellas.

Veamos el siguiente diagrama de clases ejemplo:

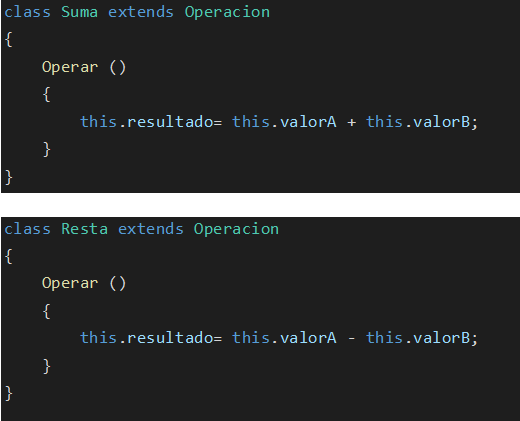


En el mismo podemos observar una clase abstracta llamada Operación la cual define 3 atributos. Luego, tenemos otras dos clases que heredan de esta: Suma y Resta. Ambas implementan el método Operar. Sin embargo, el comportamiento del método Operar deberá diferir si estamos hablando de sumas o restas.



Como podemos observar, debemos definir la clase abstracta anteponiendo el modificador “***abstract”***previo a la definición de la clase y al método que deseamos que forzosamente en las clases derivadas sea implementado.

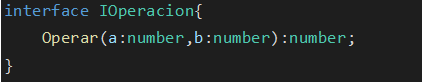
 Ejemplo:



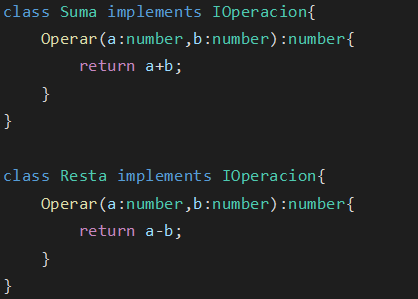
### **Polimorfismo por interfaces**

Recordemos que una interfaz es un CONTRATO por lo que define propiedades y métodos, pero no su implementación.

Las interfaces, como las clases, definen un conjunto de propiedades, métodos y eventos. Pero de forma contraria a las clases, las interfaces no proporcionan implementación. Se implementan como clases y se definen como entidades separadas de las clases. Una interfaz representa un contrato, en el cual una clase que implementa una interfaz debe implementar cualquier aspecto de dicha interfaz exactamente como esté definido.  
  
En typescript:

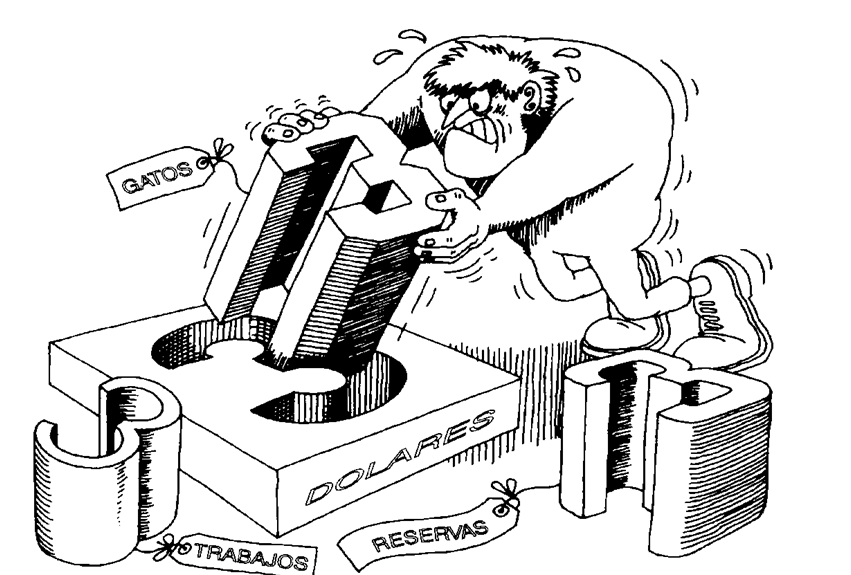


Para implementar la interfaz en nuestra clase Suma y Resta (y lograr el polimorfismo) debemos utilizar la palabra ***implements***como sigue:



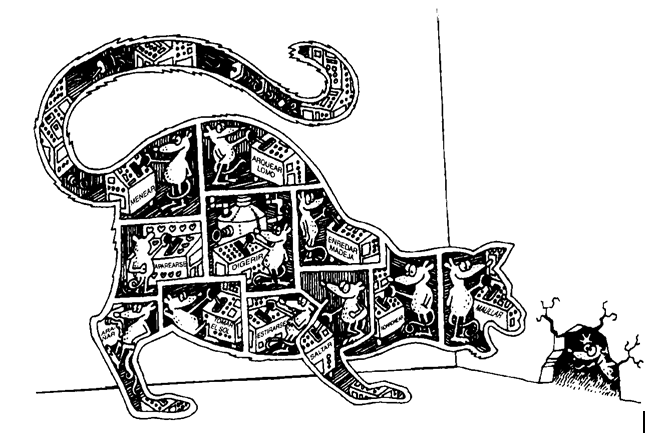
## Tipificación

Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas.



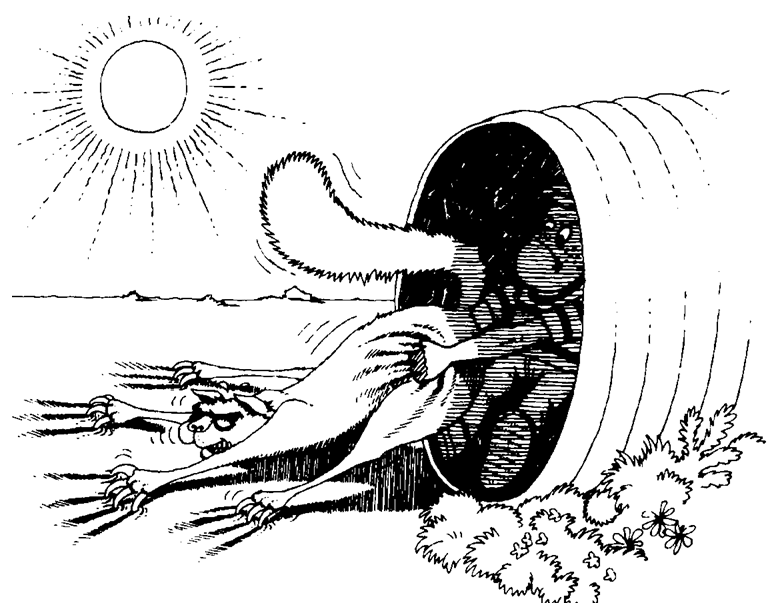
## Concurrencia

La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.

La concurrencia permite a dos objetos actuar al mismo tiempo. 

## Persistencia

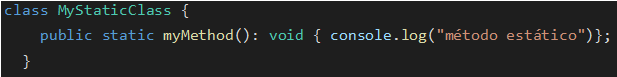
La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

Conserva el estado de un objeto en el tiempo y en el espacio. 

# **Clases estáticas**

Una static class es aquella clase que se usa sin necesidad de realizar una instanciación de la misma. Se utiliza como una unidad de organización para métodos no asociados a objetos particulares y separa datos y comportamientos que son independientes de cualquier identidad.

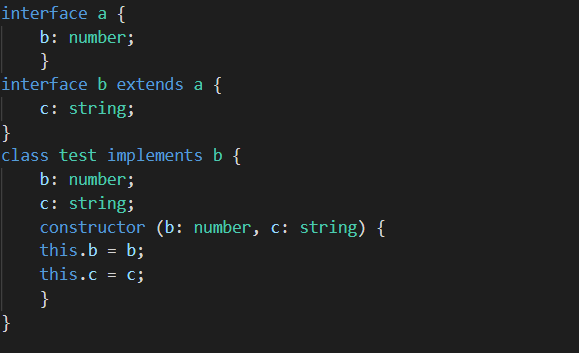
Desafortunadamente no existen clases estáticas en Typescript, aunque sí podemos definir sus métodos como estáticos y trabajar con ella sin instanciar el  objeto.



# **Interfaces**

Además de ser útiles para implementar el polimorfismo, las interfaces nos permiten crear nuevos tipos y de esta manera  comprobar los tipos de las variables. Ej cuando se pasan como argumentos.

Ejemplo:

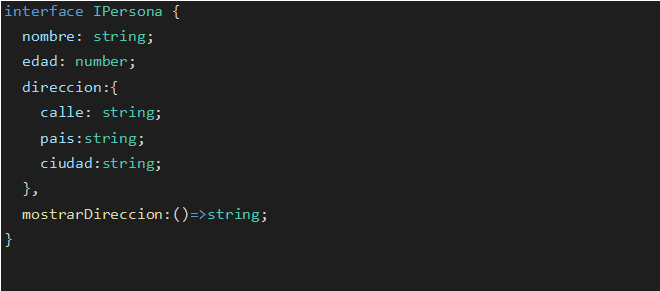


En el ejemplo anterior podemos observar que la clase test hereda de b quien a su vez, hereda de a. Entonces, test tiene acceso

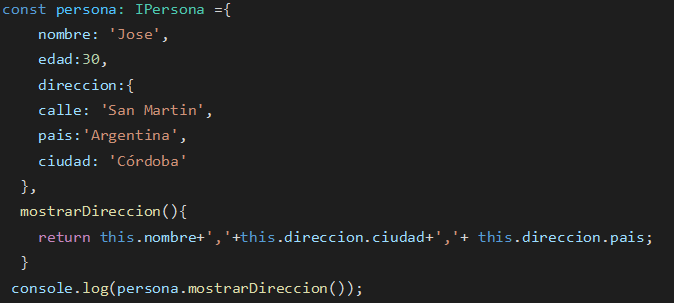
a las variables b y c las cuales no podrán ser de otro tipo que los especificados en las clases de quienes derivan.

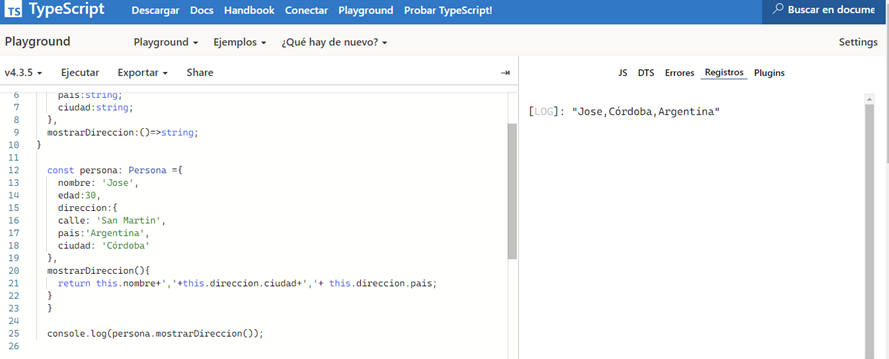
Typescripts permite crear interfaces según cuantas sean necesarias permitiéndonos evitar el anidamiento de objetos.

Ejemplo Interfaz Persona:

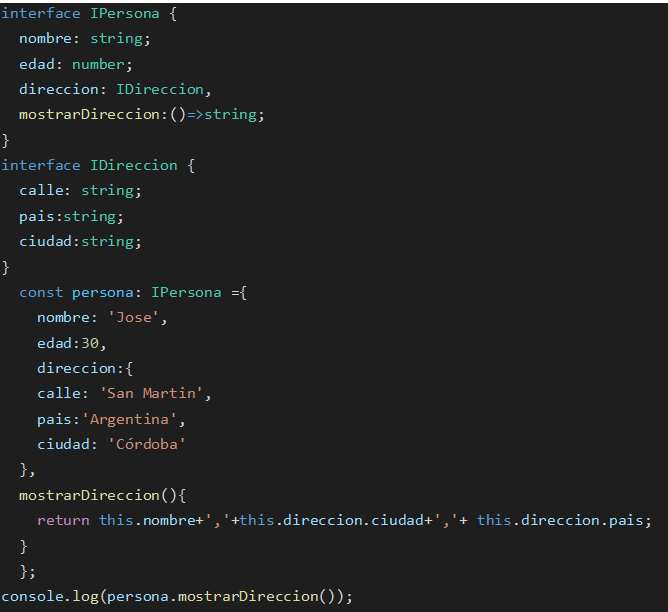


Instanciando un objeto en función de su interfaz tenemos:



De lo anterior podemos observar que, tenemos una interfaz IPersona que anida la declaración de “direccion”. 

Sin embargo, typescripts nos permite separar en más de una interfaz y así evitar el anidamiento que en algunos casos puede ser difícil de interpretar.



Como podemos observar, la interfaz IPersona está compuesta por otra IDireccion evitando así un anidamiento difícil de seguir o comprender.

Tarea desafío a subir del módulo

Te recordamos e invitamos a que utilices este conocimiento para practicar el lenguaje que utilizaras en el modulo siguiente,  así  podrás subir tus prácticas  **al repositorio de Github.**  
El objetivo es poner en práctica todo lo aprendido. Para ello deberás considerar tanto el material de lectura como los contenidos desarrollados en los encuentros en vivo y los aportes de otros compañeros.