


Ferrando Eduardo – Ferrando Matias



PROGRAMACION ORIENTADO A OBJETOS

Apunte de Catedra – Lenguaje C#

CONTENIDO

Glosario.....	3
¿Qué es un paradigma de programación?	4
Tipos de paradigmas.....	4
Paradigma de objetos.....	5
Ventajas de la POO	6
Características	7
Clases y Objetos.....	7
Componentes de una clase:	8
Atributos	8
Constructores	10
Métodos	13
Destrucción	19
Instanciación.....	20
La palabra reservada “this”	20
La Visibilidad	21
Encapsulamiento	22
Modificadores de acceso	24
• Public.....	24
• Protected	24
• Internal.....	24
• protected internal.....	24
• private.....	24
• private protected.....	24
Relaciones entre objetos	25
Relaciones Simples	25
Relaciones Múltiples.....	26
Herencia.....	26
Principio de Sustitución	32
Interfaces.....	33
Restricciones.....	34
Polimorfismo	34
Clases Anónimas	39
Clases Abstractas	40
Método Abstracto	41

Clases selladas	41
Structs (Estructuras)	42
Enum (Tipos Enumerados)	44
Programación Genérica	45

GLOSARIO

Atributos o propiedades: Características que posee una clase, describen al objeto. Por ejemplo, si la clase es “automóvil”, los atributos podrán ser el color, el modelo y la marca.

Clase: Son la representación de un conjunto de funciones (llamadas métodos) y variables (llamadas propiedades) designadas para trabajar juntas y proveer su interfaz.

Constructores: Se encargan de inicializar atributos y configurar todo lo que sea necesario antes de que comience a existir el objeto.

Encapsulamiento: Acción de reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto aumenta la cohesión de los componentes del sistema.

Herencia: Por un lado, define la relación “es un” entre dos clases (padre-hijo). Por ejemplo, la clase automóvil “es un” vehículo. Este mecanismo permite reutilizar en la clase “hijo” los métodos y propiedades de la clase “padre” de la cual hereda.

Instancia de clase: Mientras que una clase es una abstracción única, a cada objeto que tengo de esa clase se lo denomina “instancia de clase”.

Métodos: Conjunto de funcionalidades asociadas a un objeto. Por ejemplo, si el objeto es un auto, los métodos o funcionalidades serían “poner en marcha” o “avanzar”.

Objeto: En OOP, un objeto es una abstracción que sabe escuchar, responder y enviar ciertos mensajes.

OOP: En inglés, Object Oriented Programming o Programación Orientada a Objetos.

Operaciones estáticas: Son operaciones que corresponden a la clase y no a las instancias de dicha clase, es decir a los objetos.

Paradigma: Modelo o patrón aceptado por un grupo de personas. En programación, el paradigma determina cómo serán los bloques de construcción de los programas, es decir, los elementos que los constituyen.

Polimorfismo: Es la manera que existe en la OOP de tratar de la misma forma a distintos objetos.

Relaciones entre clases: Pueden ser simples (cuando una clase se relaciona únicamente con otra) o múltiples (una clase se relaciona con varias clases). Por ejemplo, auto tiene una relación simple con motor, porque un auto puede tener únicamente un motor. Sin embargo, auto tiene una relación múltiple con rueda, porque un auto puede tener varias ruedas.

¿QUÉ ES UN PARADIGMA DE PROGRAMACIÓN?

Un Paradigma es un concepto abstracto, la definición de un modelo o patrón en cualquier disciplina física. Todo paradigma nace de la filosofía de una comunidad científica que, especializada en un tema particular, decide plantear algún nuevo concepto abstracto o forma de pensar. En este caso en particular, una parte de la comunidad científica que estudiaba la computación como una rama matemática y se especializaba en ello, forja esta nueva ideología o forma de pensar que es el Paradigma de Programación Orientado a Objetos. En programación, el paradigma, nos determina cómo van a ser los bloques de construcción de los programas, es decir, los elementos que los constituyen.

TIPOS DE PARADIGMAS

- ❖ **Imperativo:** En este enfoque, los programas compuestos están por un conjunto de sentencias que indican a la computadora cómo cambiar su estado. Básicamente, se trata de secuencias de comandos que ordenan acciones específicas.
- ❖ **Declarativo:** A diferencia del paradigma imperativo, aquí los programas se centran en describir los resultados deseados sin especificar claramente los pasos necesarios para lograrlos. En lugar de decirle a la computadora qué hacer, se le dice qué se espera obtener como resultado.
- ❖ **Lógico:** En este paradigma, el problema se modela mediante enunciados de lógica de primer orden. Se utilizan reglas lógicas y restricciones para definir el comportamiento del programa y resolver problemas mediante la inferencia lógica.
- ❖ **Funcional:** En este enfoque, los programas se componen principalmente de funciones. Una función es una implementación de un comportamiento específico que toma un conjunto de datos de entrada y devuelve un valor de salida. Se enfoca en el cálculo y la transformación de datos.
- ❖ **Paradigma de Programación Orientada a Eventos:** Se centra en la programación de eventos y respuestas a esos eventos. Los programas reaccionan a eventos externos o internos, como acciones del usuario o cambios de estado, y ejecutan las respuestas correspondientes.
- ❖ **Orientado a objetos:** Aquí, el comportamiento del programa se organiza en torno a objetos. Los objetos son entidades que representan elementos del problema que se desea resolver y tienen atributos y comportamiento asociados. Los objetos interactúan entre sí a través de mensajes y colaboran para lograr los objetivos del programa.
- ❖ **Paradigma de Programación Concurrente:** Se enfoca en la programación de sistemas concurrentes, donde múltiples tareas se ejecutan simultáneamente y pueden comunicarse y sincronizarse entre sí.
- ❖ **Paradigma de Programación Basada en Componentes:** Se basa en la construcción de software mediante la composición de componentes reutilizables. Los componentes son módulos

independientes que encapsulan funcionalidades específicas y se pueden combinar para formar sistemas más grandes.

PARADIGMA DE OBJETOS

El paradigma de programación orientada a objetos plantea que todo sistema o proceso informático puede modelarse con Objetos que se encuentren vivos en algún tipo de Ambiente y se relacionan con otros Objetos enviando y recibiendo Mensajes. Es decir, programar bajo este paradigma implica que nuestros programas deberán ser pensados sólo con objetos y mensajes. Un objeto, en POO, es un ente completamente abstracto que sabe escuchar, responder y enviar ciertos mensajes. Un Mensaje es una comunicación dirigida de un objeto a otro, es decir, información enviada desde emisor hasta el receptor. El paradigma de objetos tiene otros conceptos que completan este esquema de Objeto-Mensaje.

Como todo concepto teórico o abstracto que quiera ser llevado a la vida real o a la práctica, requiere de algún tipo de implementación. Si queremos desarrollar un software, con la idea teórica o paradigma únicamente no nos alcanzaría, es necesario tener algo concreto con lo cual trabajar, una implementación del paradigma.

Cuando hablamos de una “implementación” estamos hablando de un lenguaje de programación orientado a objetos, es decir, que contempla los conceptos teóricos del paradigma orientado a objetos. Cada uno de estos lenguajes implementará cada uno de estos conceptos de forma diferente. Por eso es importante diferenciar los “conceptos del paradigma” de la “implementación del paradigma”. Es decir, por ejemplo, primero es necesario conocer la definición de un “objeto” según el paradigma, y luego conocer la implementación de un objeto para determinado lenguaje de programación.

OOP (Object Oriented Programming en inglés ó en español Programación orientada a objetos) se basa en el concepto de agrupar código y datos juntos en una misma unidad llamada clase. Este proceso es usualmente referido como encapsulamiento o información oculta. Estas clases son esencialmente la representación de un conjunto de funciones (llamadas métodos) y variables (llamadas propiedades) designadas para trabajar juntas y proveer una interfaz específica. Es importante entender que las clases son una estructura que no puede ser utilizada directamente, éstas deben ser instanciadas en objetos, los cuales podrán así interactuar con el resto de las aplicaciones. Es posible pensar en clases como un plano para crear un auto, por ejemplo, mientras el objeto es el auto en sí mismo, ya que se puede crear en una línea de producción. Podríamos decir que una clase es una fábrica de objetos.

En la POO, el estado, el comportamiento y las propiedades son conceptos relacionados pero que tienen significados ligeramente diferentes en el contexto de la programación orientada a objetos.

- ❖ **Estado:** El estado se refiere a las características o valores que describen el objeto en un momento específico. Representa la condición actual del objeto y puede cambiar a lo largo del tiempo. Por ejemplo, para un objeto "Coche", el estado puede incluir características como la velocidad actual, la posición en la carretera, el nivel de combustible, etc. El estado se almacena en los atributos o variables internas del objeto y puede modificarse mediante la ejecución de métodos.
- ❖ **Comportamiento:** El comportamiento se refiere a las acciones o funciones que un objeto puede realizar. Representa lo que un objeto puede hacer o las operaciones que puede llevar a cabo. Los comportamientos se definen mediante métodos en la programación orientada a objetos. Siguiendo el ejemplo anterior, los comportamientos de un objeto "Coche" podrían incluir métodos como "acelerar", "frenar", "girar", etc. Estos métodos permiten al objeto interactuar con su entorno y cambiar su estado.
- ❖ **Propiedades:** Las propiedades son las características distintivas o atributos que posee un objeto. Estas características describen las cualidades o aspectos del objeto. Las propiedades están relacionadas con el estado del objeto, pero se utilizan para especificar los atributos específicos que definen un objeto en particular. Por ejemplo, para un objeto "Persona", las propiedades podrían ser el nombre, la edad, la altura, etc. Estas propiedades ayudan a identificar y distinguir a cada instancia única de un objeto.

Ventajas de la POO

- **Escalabilidad:** Capacidad de crecimiento. Que un sistema sea escalable, significa que está preparado para poder crecer sin limitaciones. En casos de desarrollo corporativos como por ejemplo el de un sistema bancario, donde constantemente se necesitan agregar nuevos módulos y cambios en su comportamiento, es necesario contar con un lenguaje de programación que permita fácilmente administrar estos tipos de desarrollos empresariales.
- **Mantenimiento:** Una de las principales características del paradigma de objetos, es su capacidad realizar cambios en el programa a muy bajo costo (esfuerzo del programador). Gracias a una buena organización de las clases y sus relaciones es posible hacer pequeños pero importantes cambios que impacten luego en todas las partes del sistema.
- **Seguridad:** Este paradigma posee lo que se denomina como encapsulamiento y visibilidad, lo cual permite establecer diferentes niveles de seguridad dentro de un equipo de desarrolladores, organizando el trabajo en equipo y limitando la posible existencia de errores o incongruencias cometidos por los programadores.
- **Reutilización:** A través de la correcta utilización de jerarquía de clases, y la redefinición de funciones – polimorfismo, se puede reutilizar un importante porcentaje de código, evitando escribir subprogramas parecidos dentro de un mismo desarrollo.

- **Simplicidad:** Cuando el código de un programa desarrollado con lenguaje estructurado, crece indefinidamente, se llega a un punto donde el mismo se vuelve confuso y engorroso. El paradigma de objetos provee simplicidad al poder organizar la complejidad del mismo a través de su estructura de clases y objetos.

Características

Las siguientes son las características más importantes:

- ❖ **Abstracción:** La abstracción es el proceso de identificar las características esenciales y relevantes de un objeto en un contexto particular, ignorando los detalles irrelevantes. Permite enfocarse en los aspectos importantes de un objeto y crear representaciones más simples y manejables.
- ❖ **Encapsulación:** La encapsulación consiste en agrupar los datos y los métodos relacionados en un solo objeto, ocultando los detalles internos y teniendo una interfaz para interactuar con el objeto. Esto permite un mayor nivel de modularidad y facilita la reutilización de código.
- ❖ **Herencia:** La herencia es un mecanismo que permite que una clase herede las propiedades y comportamientos de otra clase. Permite definir nuevas clases basadas en clases existentes, aprovechando la reutilización de código y presentando relaciones jerárquicas entre las clases.
- ❖ **Polimorfismo:** El polimorfismo permite que un objeto pueda presentar múltiples formas o comportamientos. Permite que diferentes objetos respondan de manera diferente a la misma llamada de método, lo que facilita la flexibilidad y extensibilidad del código.
- ❖ **Modularidad:** La modularidad se refiere a la capacidad de dividir un sistema en módulos o componentes independientes y cohesivos. Cada módulo es responsable de una funcionalidad y específica se puede desarrollar, mantener y probar de forma separada. Esto facilita la comprensión, la colaboración y la gestión del código.
- ❖ **Reutilización de código:** La POO fomenta la reutilización de código a través de mecanismos como la herencia y la composición. Los objetos y las clases pueden ser utilizados en diferentes contextos y proyectos, lo que ahorra tiempo y esfuerzo al no tener que escribir código desde cero.

CLASES Y OBJETOS

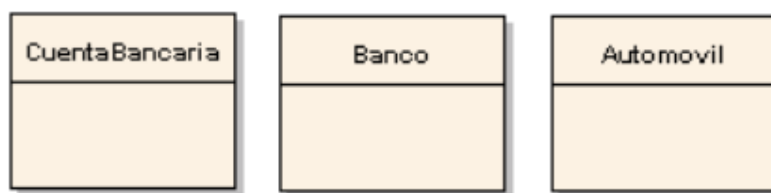
Los conceptos de clase y objeto están estrechamente relacionados, pero tienen diferencias distintivas. Un objeto es una entidad concreta que existe en el espacio y en el tiempo, mientras que una clase representa una abstracción, la esencia o la plantilla de un objeto.

En el contexto de la programación orientada a objetos, una clase se define como un conjunto de objetos que comparten una estructura común y un comportamiento común. Un objeto, por otro lado, es una instancia

específica de una clase. Los objetos que comparten estructura y comportamientos similares pueden agruparse en una clase. En este paradigma, los objetos siempre pertenecen a una clase, de la cual heredan su estructura y comportamiento. Aunque durante mucho tiempo se ha utilizado incorrectamente el término "objeto" para referirse a las clases, es importante resaltar que son conceptos distintos.

Las clases están compuestas principalmente por atributos (variables) y métodos (procedimientos o rutinas). En la programación orientada a objetos, todo se realiza a través de clases y objetos. Un objeto es una combinación de atributos (estado) y operaciones (comportamiento). Los objetos deben ser vistos como abstracciones de la realidad con características representativas. Representan ideas del mundo real, como animales o personas. Las clases, por otro lado, son plantillas que contienen características y se consideran los moldes para los objetos. Las clases se suelen denominar como sustantivos en singular, como "CuentaCorriente" o "Automóvil".

Si tenemos una clase "CuentaBancaria", podemos tener infinita cantidad de objetos de "CuentaBancaria", pero cada una tendrá sus valores individuales, aunque todas pertenezcan a esta clase. Decimos que un objeto pertenece a una clase, cuando es una instancia de esa clase; es decir, al crear un objeto de la clase "Banco" y asignarle el nombre "bancoNacion", el objeto "bancoNacion" es una instancia de "Banco". Continuando con este ejemplo, podríamos crear más instancias de "Banco" y asignarles nombres como "bancoFrances" o "bancoProvincia", y todos ellos tendrían algo en común, que pertenecen a la clase "Banco", a pesar de que cada objeto tenga sus distintos valores, como por ejemplo, distinta cantidad de clientes, de sucursales, etc (representados por los atributos). A continuación, se muestran algunas clases representadas gráficamente:

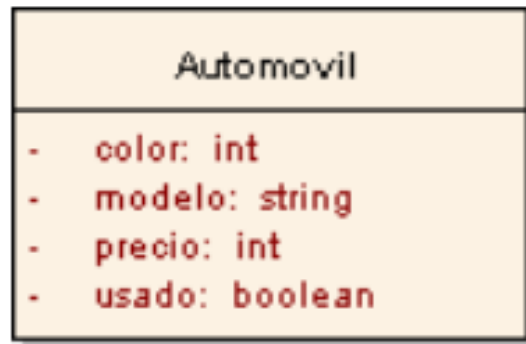


COMPONENTES DE UNA CLASE:

Atributos

Los atributos son las características individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades. Los atributos se guardan en variables denominadas de instancia, y cada objeto particular puede tener valores distintos para estas variables.

Como mencionamos anteriormente, los atributos son las variables contenidas por los objetos. Las clases definen los atributos y los objetos los “completan”. Las variables de una clase definen las características de sus objetos, por ejemplo:



En este caso creamos una clase “Automovil” y colocamos atributos color, modelo, precio, y usado que, al adquirir valores en un objeto, lo diferenciarán a este de otras instancias (objetos) de la clase “Automovil”. Lo mismo podríamos hacer con una clase “CuentaBancaria”, como se muestra a continuación:



En este caso la clase “CuentaBancaria” cuenta con un atributo “saldo”. En resumen, los atributos son las características de una clase.

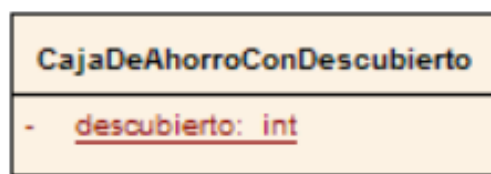
Podemos encontrar la definición de atributo, mediante la cual definimos que un objeto tiene un conjunto de “características” que lo describe, y que se definen en una clase. Por ejemplo, el atributo nombre, va a ser una palabra, es decir, va a ser un String; de la misma manera, la edad, es un número, es decir, va a ser un “int”. “int” y “String” son tipos. Un tipo es la forma de describir o almacenar un dato. Cuando decimos que tenemos un dato del tipo “int”, estamos diciendo que podría ser cualquier número de tipo entero. De la misma manera, un objeto Persona podría tener un atributo “Trabajo”, donde trabajo es también un objeto con sus propias características, es decir, el trabajo podría tener un empleador, un nombre, una ubicación, etc. De esta manera, estamos diciendo que Persona, tiene como atributo “Trabajo”, ¿qué será de qué tipo? De tipo Trabajo! Acá empezamos a ver la potencia de los objetos.

Anteriormente, nosotros trabajamos con un conjunto acotado de “tipos” para las variables (int, String, boolean, etc), ahora, con los objetos, nosotros podemos crear nuestros propios “tipos”. Cualquier clase que nosotros creamos, de ahora en más, eso va a ser un nuevo tipo con el que podemos trabajar. De la misma

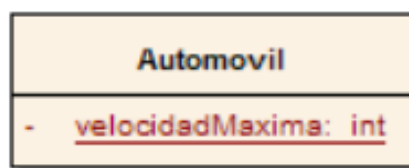
manera si creamos una clase “Banco”, de ahora en más vamos a tener objetos que son instancias de la clase Banco, o también podemos decir que tenemos objetos del tipo Banco.

Atributos de Clase

También llamados atributos estáticos, son atributos que pertenecen a la clase y no a un objeto o instancia de clase. Esto significa que son atributos compartidos por todos los objetos. Por ejemplo, en una clase “CajaDeAhorroConDescubierto” podríamos declarar un atributo estático llamado “descubierto” que contenga el valor en pesos de descubierto que puede brindar esa caja de ahorro a los clientes, es decir, no necesitamos que cada instancia de “CajaDeAhorroConDescubierto” almacene un valor independiente, sino tan solo uno que sirve para todas las instancias. Este es el tipo de usos que se le da a los atributos estáticos.



En el diagrama anterior el atributo “descubierto” aparece subrayado, esto es lo que nos indica que es un atributo estático. Lo mismo sucedería para una clase “Automovil” con un atributo estático “velocidadMaxima”:



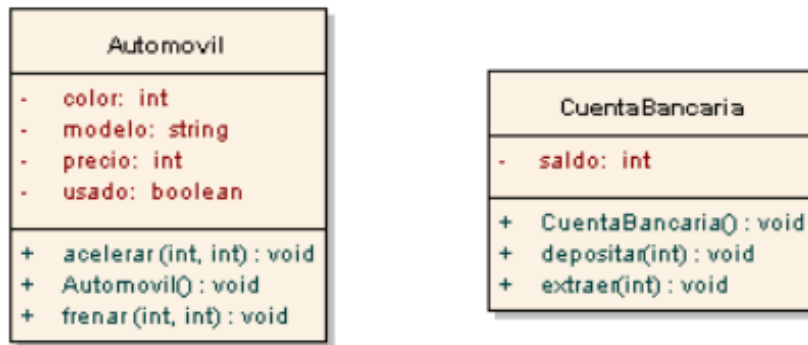
En resumen, los atributos de clase (o estáticos) son atributos que existen en la clase y no en cada uno de los objetos, motivo por el cual son compartidos por todos los objetos.

Constructores

Toda clase tiene un constructor. Los constructores son operaciones que son invocadas al instanciar una clase. Los constructores se encargan de inicializar atributos y configurar todo lo que sea necesario antes de que comience a existir el objeto.

Al tratarse de operaciones, los constructores pueden contener cualquier comportamiento, aunque éstas no tengan que ver exactamente con “construir” el objeto. Es posible tener más de un constructor en una misma clase; en este caso, cada constructor debe aceptar parámetros distintos para poder así distinguirlos.

Los constructores dan un estado inicial a las propiedades. SIEMPRE debe tener el mismo nombre de la clase.



Por lo tanto la clase “Automóvil” tendrá un constructor llamado “Automovil()”, y la clase “CuentaBancaria” tendrá un constructor llamado “CuentaBancaria()”. En los ejemplos anteriores los constructores no tienen parámetros. En resumen, un constructor es una operación que es invocada al instanciar una clase.

Cuando el Constructor se omite, en el caso de C#, el compilador crea un constructor vacío (al cual no podemos acceder ni ver). Cuando instanciamos una clase creando el/los Objeto/s de dicha clase, hacemos referencia al constructor.

Ejemplo:

```
Auto InstanciaAuto = new Auto();
```

Aquí estamos referenciando al Constructor
De la clase.

Una sobrecarga de constructores es cuando hay más de un constructor. Por ejemplo, uno sin recibir parámetros y otro recibiendo parámetros.

Ejemplo:

```
Static void main (String [] Args)
{
    // Se crea una instancia de la clase Auto
    Auto InstanciaAuto = new Auto();    // Se ejecuta automáticamente el constructor
    LblRuedas.Text = Convert.ToString(InstanciaAuto.getRuedas());
}
Class Auto
{
    private int ruedas;
    private int largo;
    private int ancho;
    private bool climatizador;
    private string tapizado;

    public Auto()
    {
```

```

        ruedas = 4;
        largo = 5000;
        ancho = 1500;
    }
    public int getRuedas()
    {
        // Este método permite que podamos tomar un valor (Método Getter)
        return ruedas;    // El return devuelve un valor o sale del método
    }
}

```

Así cómo es posible tener más de un constructor (que es una operación), es posible tener más de una operación con el mismo nombre dentro de la misma clase. Para poder hacer esto es necesario que se cumplan las mismas condiciones que para los constructores, es decir, que cada sobrecarga de operación tenga distintos tipos de parámetros. ¿Para qué podría servir tener varias operaciones con el mismo nombre, pero con distintos parámetros? Por ejemplo, si quisiéramos crear una operación llamada "extraer()" que extraiga un determinado saldo de una cuenta, podríamos querer utilizarla sin determinar un tipo de moneda (utilizando la moneda del país como "por defecto"), o determinando un tipo de moneda para hacer la extracción de ese tipo de moneda (euro, dólar, etc). La sobrecarga de operaciones nos permite utilizar el mismo nombre de operación para dos operaciones distintas, es decir, que aceptan distintos parámetros y tienen distinta funcionalidad. Sin la existencia de este mecanismo, se debería crear una operación distinta por cada "versión" de nuestra operación. Sin la sobrecarga de operaciones, las operaciones están limitadas a los parámetros de una sola definición de operación.

CuentaCorriente	
-	saldo: int
+	extraer(int) : int
+	extraer(int, string) : int

Si quisiéramos aplicar la sobrecarga de operaciones en una clase “Automóvil”:

Automovil	
-	velocidad: int
+	acelerar(int) : void
+	acelerar(int, boolean) : void

La operación “acelerar()” tiene dos definiciones, en una acepta como argumento la cantidad de tiempo que debe acelerar, y en la otra la cantidad de tiempo que debe acelerar y si el auto utiliza turbo o no, representado por el segundo parámetro de tipo booleano. En resumen, la sobrecarga de operaciones es la capacidad de poder crear comportamientos distintos para la misma operación aceptando distintos tipos de parámetros.

Métodos

Un método es una subrutina cuyo código es definido en una clase y puede pertenecer tanto a una clase, como es el caso de los métodos de clase o estáticos, como a un objeto, como es el caso de los métodos de instancia.

Las operaciones son acciones contenidas por el objeto que definen su comportamiento.

Automovil	
-	color: int
-	modelo: string
-	precio: int
-	usado: boolean
+	acelerar(int, int) : void
+	frenar(int, int) : void

En este caso hemos agregado dos operaciones a la clase “Automovil”. Estas operaciones le permiten acelerar o frenar, como indican sus respectivos nombres (cuanto más autoexplicativos sean los nombres, mejor). Las operaciones son, en el 99% de los casos, verbos. Las operaciones pueden admitir parámetros. Los parámetros son variables que utilizará la operación para generar cierto comportamiento. Las operaciones acelerar y frenar tienen como parámetros dos valores “int” (números enteros) que representan la velocidad y el tiempo que acelerará o frenará.

Las operaciones de una clase “CuentaBancaria” podrían ser las siguientes:

CuentaBancaria	
-	saldo: int
+	depositar(int): void
+	extraer(int): void

Las operaciones “depositar” y “extraer” tienen un solo parámetro, este parámetro representa el monto a extraer o depositar. En resumen, las operaciones son acciones contenidas por un objeto que definen su comportamiento.

Métodos estáticos y de instancia

También llamadas operaciones estáticas, son operaciones que corresponden a la clase y no a las instancias de dicha clase, es decir a los objetos. La explicación es análoga a la presentada en el tema anterior de atributos de clase.

En los siguientes ejemplos se aplican operaciones estáticas:

CajaDeAhorroConDescubierto	Automovil
- <u>descubierto: int</u>	- <u>velocidadMaxima: int</u>
+ <u>leerDescubierto(): int</u>	+ <u>leerVelocidadMaxima(): int</u>
+ <u>modificarDescubierto(int): void</u>	+ <u>modificarVelocidadMaxima(int): void</u>

Se debe observar que en ambas clases se dispone de un atributo estático y dos operaciones estáticas, y en este caso ambas operaciones estáticas se utilizan para modificar los atributos estáticos. En resumen, las operaciones de clase (o estáticas) son operaciones que existen en la clase y no en cada uno de los objetos, motivo por el cual son compartidos por todos los objetos.

En resumen, un método declarado con un modificador static es un *método estático*. Un método estático no opera en una instancia específica y solo puede acceder directamente a miembros estáticos.

Un método declarado sin un modificador static es un *método de instancia*. Un método de instancia opera en una instancia específica y puede acceder a miembros estáticos y de instancia. Se puede acceder explícitamente a la instancia en la que se invoca un método de instancia como this. Es un error hacer referencia a this en un método estático. This se utiliza en un método de instancia para referirse a la instancia real y acceder a sus

atributos y métodos. Sin embargo, no se puede utilizar `this` en un método estático, ya que no hay una instancia específica en la que invocar el método.

Ejemplo:

La siguiente clase `Entity` tiene miembros estáticos y de instancia.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity()
    {
        serialNo = nextSerialNo++;
    }
    public int GetSerialNo()
    {
        return serialNo;
    }
    public static int GetNextSerialNo()
    {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value)
    {
        nextSerialNo = value;
    }
}
```

Cada vez que se crea una nueva instancia de la clase "Entity", el constructor se ejecuta y asigna a la instancia un número de serie único. El número de serie se obtiene del campo estático "nextSerialNo", y luego se incrementa para la siguiente instancia.

El método de instancia "GetSerialNo()" se utiliza para obtener el número de serie de una instancia específica de "Entity". Simplemente devuelve el valor del campo de instancia "serialNo".

Los métodos estáticos "GetNextSerialNo()" y "SetNextSerialNo(int value)" se utilizan para obtener y establecer el valor del campo estático "nextSerialNo". Estos métodos pueden acceder al campo estático directamente porque pertenecen a la clase en su conjunto, no a una instancia específica. Esto significa que todos los objetos "Entity" comparten el mismo campo estático "nextSerialNo".

Es importante destacar que los métodos estáticos no pueden acceder directamente al campo de instancia "serialNo". Esto se debe a que los métodos estáticos no tienen acceso a los campos de instancia individuales,

ya que no están asociados con una instancia en particular. Sin embargo, pueden acceder al campo estático "nextSerialNo", que es compartido por todas las instancias de la clase.

Accesors: Métodos de Acceso Getter y Setter

Los **Setters** y **Getters** son métodos de acceso, lo que indica que son siempre declarados públicos, y nos sirven para dos cosas:

Setter

Del Inglés Set, que significa establecer, pues nos sirve para asignar un valor inicial a un atributo, pero de forma explícita, además el Setter nunca retorna nada (Siempre es void), y solo nos permite dar acceso público a ciertos atributos que deseemos que el usuario pueda modificar. Cuando los parámetros del método setter se llaman igual a las variables de clase se utiliza la palabra reservada "**this**" para hacer referencia al campo de clase. Un ejemplo de un método setter seria el siguiente:

```
Private int valor;  
public void setExtras (int parametro)  
{  
    If (parámetro > 0)  
    {  
        valor = parámetro;  
    }  
}
```

Getters

Del Inglés Get, que significa obtener, pues nos sirve para obtener (recuperar o acceder) el valor ya asignado a un atributo y utilizarlo para cierto método. Siempre va a llevar dentro un "**return**", ya que debe devolver un valor.

```
public int getExtras ()  
{  
    Return valor;  
}
```

Properties (Propiedades)

Las propiedades en la programación permiten acceder a los valores de una manera similar a como se accedería a un campo público, pero con la ventaja de poder controlar y aplicar reglas específicas. Las propiedades nos brindan una capa de abstracción que evita violaciones no deseadas en nuestras reglas de negocio.

Una propiedad puede tener tanto un método "get" como un método "set", lo que permite la lectura y escritura de su valor. Sin embargo, no todas las propiedades tienen que ser de lectura y escritura. Es posible definir una propiedad de solo lectura, que solo tiene un método "get" para obtener su valor, o una propiedad de solo escritura, que solo tiene un método "set" para asignar su valor.

Al proporcionar métodos "get" y "set" personalizados para una propiedad, podemos controlar cómo se accede y se modifica el valor de esa propiedad. Podemos aplicar lógica adicional dentro de estos métodos para realizar validaciones, restricciones o cálculos antes de permitir o aplicar un nuevo valor. Esto nos permite mantener la integridad de los datos y garantizar que se cumplan las reglas definidas en nuestro código.

Sintaxis: `public double Salario { get; set; }`

Ejemplo:

```

Class empleado
{
    private double Salario;
    private double EvaluaSalario (double Salario)
    {
        if (Salario < 0) return 0;
        else return Salario;
    }
    public double SALARIO
    {
        get
        {
            return this.Salario;
        }
        set
        {
            this.Salario = EvaluaSalario(value);
        }

        /*
        También la sintaxis puede ser la siguiente:
        get => this.Salario;
        set => this.Salario = EvaluaSalario(value);
        */
    }
}

Static void main (String [] Args)
{
    empleado juan = new empleado();
    juan.SALARIO = 15000;           // Se aplica el método Set
    Console.WriteLine(juan.SALARIO); // Se aplica el método Get
}

```

Lambda

Una expresión lambda es una forma concisa de definir una función anónima en la programación. Se utiliza para simplificar y reducir la cantidad de código necesario al trabajar con funciones o expresiones que se utilizan de manera repetitiva.

En el siguiente ejemplo, se muestra una propiedad "SALARIO" con sus respectivos métodos de acceso "get" y "set". En lugar de utilizar bloques de código completos para definir los métodos "get" y "set", se utilizan expresiones lambda para simplificar la implementación.

La expresión lambda "Get => Return this.Salario;" se utiliza para definir el método "get" de la propiedad "SALARIO". En este caso, la expresión lambda simplemente devuelve el valor de la variable "Salario" de la instancia actual.

La expresión lambda "Set => This.Salario = EvaluaSalario(Value);" se utiliza para definir el método "set" de la propiedad "SALARIO". Aquí, la expresión lambda asigna el valor proporcionado ("Value") a la variable "Salario" de la instancia real después de evaluarlo utilizando la función "EvaluaSalario".

Al utilizar expresiones lambda, se logra una sintaxis más concisa y legible. En lugar de escribir bloques de código completos con llaves y palabras clave como "return", se utiliza la flecha (=>) para indicar el flujo de la función y realizar las operaciones necesarias de forma más compacta.

Ejemplo:

```
Public double SALARIO
{
    get => this.Salario;
    set => this.Salario = EvaluaSalario(value);
}
```

Métodos virtuales y de Reemplazo

Cuando se utiliza el modificador "virtual" en la declaración de un método de instancia en C#, se indica que el método es un método virtual. Por otro lado, si no se especifica el modificador "virtual", el método se considera no virtual.

Cuando se invoca un método virtual, la implementación del método que se ejecutará se determina según el tipo en tiempo de ejecución de la instancia en la que se realiza la invocación. Esto significa que si se tiene una clase base con un método virtual y se crea una instancia de una clase derivada, al invocar el método virtual, se confirma la implementación demostrada en la clase derivada. La resolución de la implementación del método se realiza en tiempo de ejecución.

En contraste, en una invocación de método no virtual, la implementación del método se determina según el tipo en tiempo de compilación de la instancia. Esto significa que se sigue la implementación del método tal como se ha definido en la clase base, incluso si se invoca el método en una instancia de una clase derivada.

Un método virtual puede ser reemplazado en una clase derivada utilizando el modificador "override". Al utilizar "override" en la declaración de un método de instancia en una clase derivada, se especifica que dicho método está reemplazando a un método virtual heredado de la clase base con la misma firma (nombre y parámetros). El método de reemplazo proporciona una nueva implementación del método virtual heredado, especializando su comportamiento según las necesidades de la clase derivada.

Un ejemplo de esto se verá en el apartado de Polimorfismo.

Destruyores

Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase. El destructor realizará procesos necesarios cuando un objeto termine su ámbito temporal, por ejemplo, liberando la memoria dinámica utilizada por dicho objeto o liberando recursos usados, como archivos, dispositivos, etc.

Los destructores tienen ciertas características:

- ❖ También tienen el mismo nombre que la clase a la que pertenecen, pero tienen el símbolo ~ delante.
- ❖ No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- ❖ No tienen parámetros.
- ❖ No pueden ser heredados.
- ❖ Deben ser públicos, no tendría ningún sentido declarar un destructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.
- ❖ No pueden ser sobrecargados, lo cual es lógico, puesto que no tienen valor de retorno ni parámetros, no hay posibilidad de sobrecarga.
- ❖ Una clase no puede tener más de un destructor. Además, si se omite el destructor, el compilador lo crea automáticamente.

El sistema en tiempo de ejecución llama automáticamente a un destructor de la clase cuando la instancia de esa clase cae fuera de alcance o cuando la instancia se borra explícitamente.

El Destructor, se ejecuta en el instante que se destruye el objeto referenciado de una clase, al igual que el constructor, debe llevar el mismo nombre de la clase, pero sin ningún tipo de modificador, solo va el nombre precedido del símbolo: ~ y proseguido de los paréntesis () Ej.:

```

Public Class Alumno
{
.....
~Alumno()
{
// Aquí ponemos todo lo que queramos que se cierre al cerrar
la clase.
}
}

```

INSTANCIACIÓN

La instancia de clases, es cuando hacemos que dos clases interactúen entre sí, donde enviamos valores de una clase hacia otra, donde se reciben y se les asigna un nuevo espacio en memoria dentro de un constructor.

Cuando utilizamos POO, podemos acceder a variables que se encuentran en otras clases, solo si creamos instancias entre clases.

Las instancias de clases se crean mediante el operador new, que asigna memoria para una nueva instancia, invoca un constructor para inicializar la instancia y devuelve una referencia a la instancia. Las instrucciones siguientes crean dos objetos Point y almacenan las referencias en esos objetos en dos variables:

```

Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);

```

La memoria ocupada por un objeto se reclama automáticamente cuando el objeto ya no es accesible. No es necesario ni posible desasignar explícitamente objetos en C#.

LA PALABRA RESERVADA “THIS”

Cuando accedemos a variables de instancia dentro de una clase, puede ocurrir que existan variables con el mismo nombre tanto en la clase en la que estamos trabajando como en la clase desde la cual estamos accediendo. Esto puede generar ambigüedad y dificultar la distinción entre las variables locales y las variables de instancia.

En este escenario, la palabra clave "this" se utiliza para resolver la ambigüedad y referirse claramente a las variables de instancia de la clase actual. Al utilizar "this", estamos indicando que queremos acceder a las variables de instancia de la clase en la que nos encontramos, y no a las variables locales que pueden tener el mismo nombre.

Por ejemplo, considere el siguiente código:

```
class MiClase{
    private string nombre; // Variable de instancia
    public void EstablecerNombre(string nombre)
    {
        this.nombre = nombre; // Utilizando "this" para referirse a la variable de instancia
    }
}
```

En este caso, la clase `MiClase` tiene una variable de instancia llamada `nombre`. Dentro del método `EstablecerNombre`, se recibe un parámetro llamado `nombre` que tiene el mismo nombre que la variable de instancia. Para asignar el valor del parámetro a la variable de instancia, se utiliza `this.nombre`, lo cual deja claro que se está accediendo a la variable de instancia de la clase.

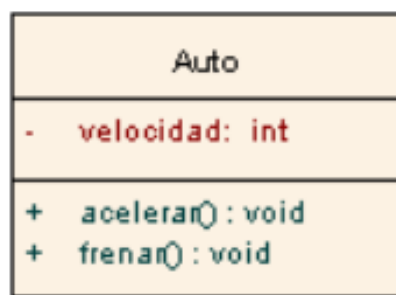
Sin el uso de `"this"`, el compilador podría interpretar que se está haciendo referencia a la variable local `nombre` del método en lugar de la variable de instancia. Al utilizar `"this"`, se evita la ambigüedad y se indica claramente que queremos acceder a la variable de instancia.

LA VISIBILIDAD

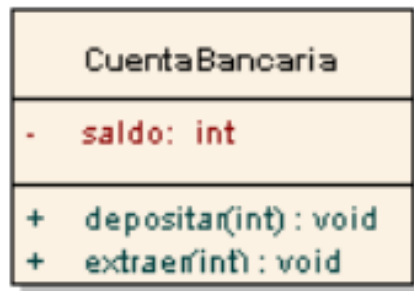
La visibilidad o accesibilidad determina el alcance que tiene un atributo u operación. El alcance (scope) de un miembro representa qué partes pueden acceder a dicho miembro. El alcance está determinado por los modificadores de visibilidad, presentados como “público” (su simbología es el signo `+`) y como “privado” (su simbología es el signo `-`)

Que un atributo u operación sea privado significa que aquel atributo u operación solo puede ser accedido desde dentro de la clase que lo contiene. En cambio, si este atributo u operación fuera público podría ser accedido por cualquier objeto.

En el siguiente ejemplo podemos ver los miembros públicos y privados de la clase “Auto” diferenciados por los signos “-” y “+”:



Esto mismo se puede aplicar con la clase “CuentaBancaria”:



Una de las dudas que surgen a la hora de entender los modificadores de acceso es “¿por qué debería de tener un miembro privado?”. Hacer que un atributo sea privado se utiliza para tratar con datos o valores los cuales no queremos que sean accedidos más que por el objeto contenedor. Las operaciones privadas cobran sentido como procesamientos internos del objeto que no queremos sean accedidas por otros objetos (tanto porque no tiene sentido que la accedan otros objetos como porque podría resultar peligroso para la integridad del objeto contenedor). El porqué de utilizar miembros privados se aclarará a continuación junto con el tema “Encapsulamiento”.

En resumen, la visibilidad determina el alcance (scope) del atributo u operación en cuestión.

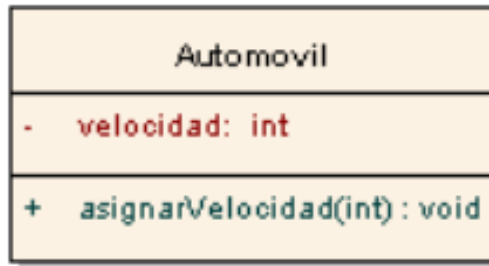
ENCAPSULAMIENTO

Conceptos previos:

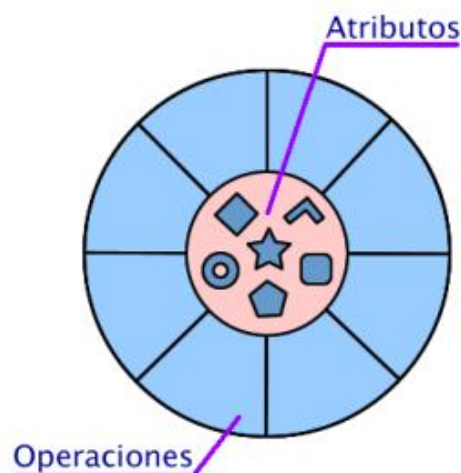
- Los **atributos**, son las variables que contiene nuestra clase.
- Los **métodos**, son las funciones que va a realizar nuestra clase.

El encapsulamiento consiste en controlar el acceso a los datos que conforman un objeto o instancia de una clase. Es decir, indicar que métodos y atributos son públicos, para poder revisar su contenido e incluso ser modificados. Y a su vez indicar que métodos y atributos son privados, para evitar el acceso a su contenido o que se realice alguna modificación en ellos.

El encapsulamiento es una de las características representativas de la POO. Este concepto tiene que ver con la forma correcta de pensar en clases y objetos. Significa aislar o independizar las clases de nuestro programa de tal manera que su funcionamiento sea independiente del de las otras clases.

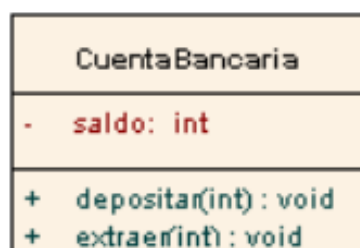


La clase “Automóvil” posee un atributo privado que representa la velocidad y posee una operación pública que le permite interactuar con otros objetos. Esta operación recibe como parámetro un valor que representa la velocidad que se le quiere asignar a dicho automovil. Si esta operación recibiera un valor negativo, podría pasarlo a un número positivo, mantener la velocidad anterior o hacer cualquier otra cosa evitando que aparezcan errores en el objeto por un dato inválido (velocidad negativa). Si en vez de utilizar un atributo privado y una operación pública, utilizáramos directamente un atributo público, este tipo de casos no podrían ser manejados y se atentaría contra la integridad del objeto. La ocultación puede visualizarse mejor en el siguiente gráfico:



El gráfico representa que los atributos están encerrados por operaciones, es decir que los atributos son privados y las operaciones son públicas. Esto significa que para poder acceder a un atributo es necesario utilizar una operación, es decir que no hay posible acceso al atributo sino no se utiliza una operación.

A continuación, se presenta el encapsulamiento con la clase “CuentaBancaria”:



Para poder modificar el atributo saldo, es necesario hacerlo a través de las operaciones depositar o extraer.

En conclusión, el encapsulamiento consiste en ocultar los atributos y métodos de una clase, para evitar que sean modificados desde otra clase. Esto es con el fin de que cuando otro programador utilice nuestra clase, no pueda cambiar su estado o comportamiento de manera imprevista o incontrolada.

La encapsulación se consigue añadiendo modificadores de acceso en las definiciones de miembros y tipos de datos.

Modificadores de acceso

Para poder realizar la encapsulación, es necesario utilizar los modificadores de acceso. Estos permiten dar un nivel de seguridad mayor a nuestras aplicaciones restringiendo el acceso a diferentes atributos, métodos o constructores asegurándonos que el usuario deba seguir una “ruta” especificada por nosotros para acceder a la información.

Cada miembro de una clase tiene asociada una accesibilidad, que controla las regiones del texto del programa que pueden tener acceso al miembro. Existen seis formas de accesibilidad posibles que se resumen a continuación.

- **Public**
Acceso no limitado.
- **Protected**
Acceso limitado a esta clase o a las clases derivadas de esta clase.
- **Internal**
Acceso limitado al ensamblado actual (.exe, .dll, etc.).
- **protected internal**
Acceso limitado a la clase contenedora, las clases derivadas de la clase contenedora, o bien las clases dentro del mismo ensamblado.
- **private**
Acceso limitado a esta clase.
- **private protected**
Acceso limitado a la clase contenedora o las clases derivadas del tipo contenedor con el mismo ensamblado.

Convenciones

Los identificadores “**public**” deben comenzar con mayúscula. Ejemplo: `public double CalcularArea();`

Los identificadores “**private**” deben comenzar con minúscula. Ejemplo: `private double valorPi;`

La diferencia entre poner publica una variable para modificarla desde otra clase y crear un método público para modificar ese valor privado radica en que en el método uno puede realizar ciertas validaciones por seguridad y quien quiere cambiar ese valor no está enterado de qué manera se realiza.

En conclusión, la manera correcta de programar en POO, sería hacer todos los Atributos (variables) privados, y colocar Métodos públicos para modificar sus valores.

RELACIONES ENTRE OBJETOS

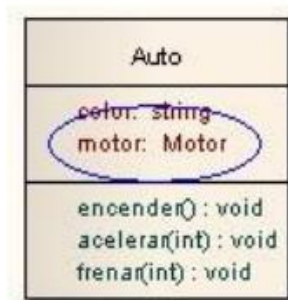
RELACIONES SIMPLES

Se produce cuando una clase se relaciona con otra clase. La relación es única Por ejemplo: Auto tiene una relación simple con Motor, porque un auto puede tener un motor únicamente

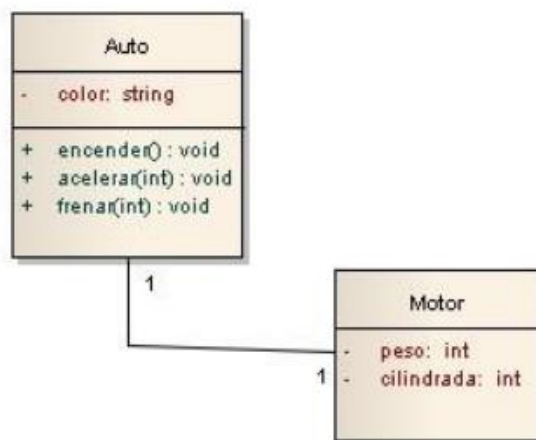
Se puede presentar como: “... un Auto tiene un Motor ...”

Existen dos formas de representar relaciones simples gráficamente:

CASO #1



CASO #2



RELACIONES MÚLTIPLES

Se produce cuando una clase se relaciona con una o muchas otras clases. La relación es múltiple, por ejemplo: Auto tiene una relación múltiple con Rueda, porque un auto puede tener varias ruedas.

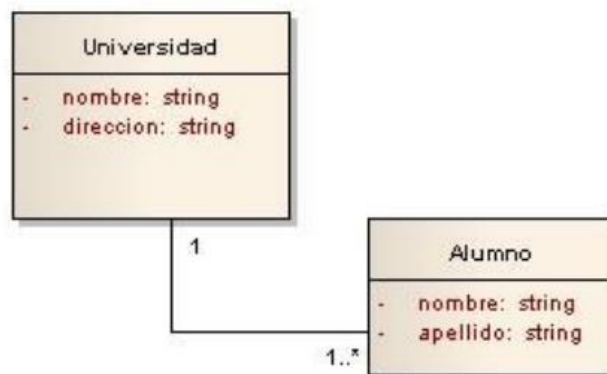
Se puede presentar como: "... un Auto tiene de una a muchas Rueda(s) ..." o "... una Universidad tiene de uno a muchos Alumno(s) ...". Se dice que una Universidad tiene una colección de alumnos.

Existen dos formas de representar relaciones simples gráficamente:

CASO #1



CASO #2

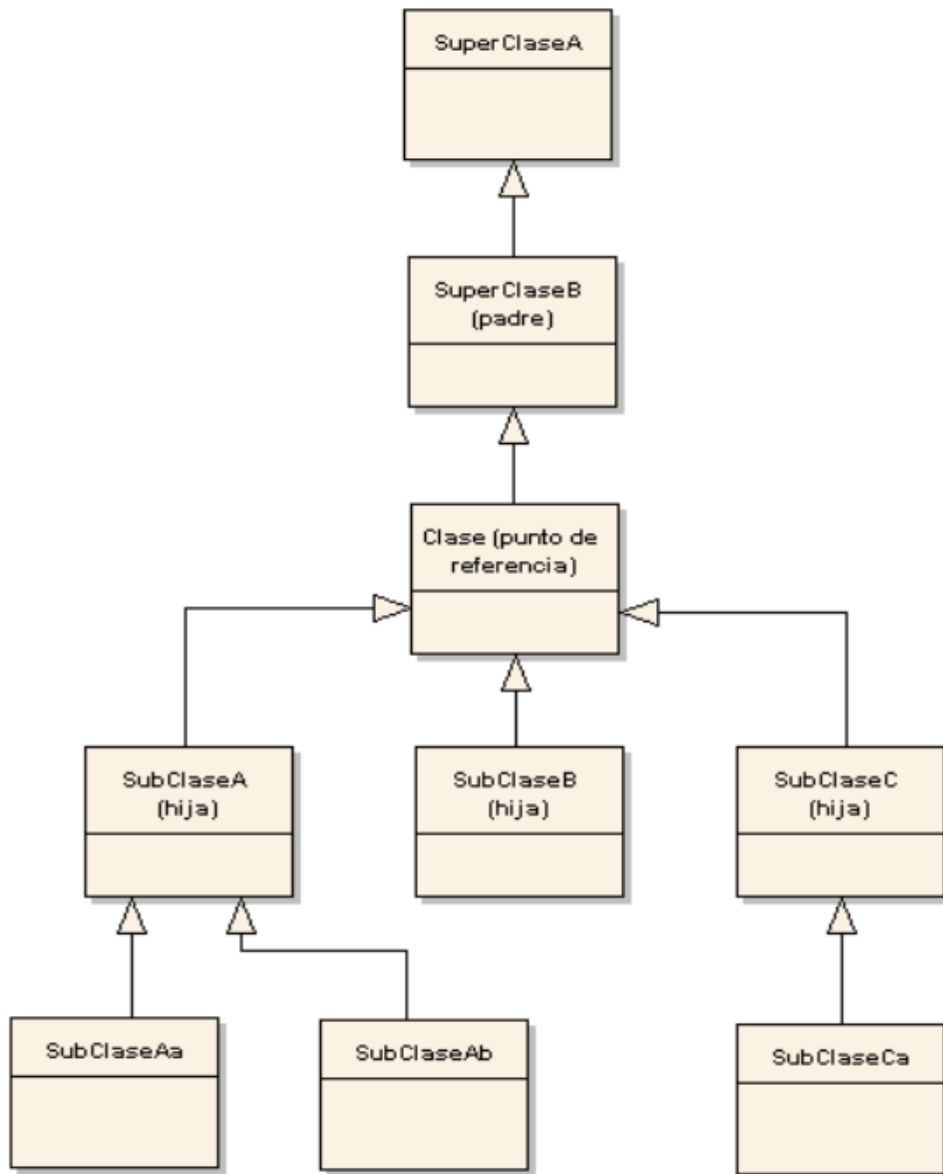


HERENCIA

La herencia es el procedimiento utilizado para reutilizar código cuando creamos nuevas clases. Lo único que se hace, es indicar al programa que queremos utilizar las variables y funciones de una clase que ya hemos creado anteriormente. Es decir, podemos utilizar los métodos y atributos que no estén declarados como privados de una clase que ya existe, y colocarlos dentro de una nueva clase sin la necesidad de volver a escribir el código.

El mecanismo de herencia es otro de los conceptos clave de la POO, y uno de sus puntos fuertes. Este mecanismo nos permite que una clase "herede de otra clase" o "extienda otra clase" recibiendo o heredando

todos los atributos y operaciones de su clase "padre". En el siguiente diagrama podemos ver cómo está compuesto un típico árbol de herencia simple:



En el diagrama anterior vemos como una clase puede tener tanto “superclases” como “subclases”. Esto significa que una clase hereda de una “superclase” o clase padre, y es heredada por “subclases” o clases hijas. La flecha indica “quien hereda a quien”, es decir, el origen de la flecha es la clase hija y el destino de la flecha es la clase padre.

Se considera clases hijas a las clases que heredan directamente de una clase y subclases tanto a las clases hijas como al resto de clases que tengan menor jerarquía en el árbol de herencia; se considera clase padre a la clase de la que se hereda directamente.

Cuando una clase hereda de otra, todas las operaciones y atributos pasan a estar disponibles en la clase hija, es decir que partimos de todos estos elementos e incluso podemos agregar nuevos o modificar estos ya heredados. Con las clases heredadas ampliamos la descripción de un concepto de la vida real, lo volvemos más específico.

La herencia o reutilización de código, es una gran ventaja porque ayuda al programador a ahorrar código y tiempo, al no tener que volver a escribir las mismas líneas de código nuevamente.

Cuando utilizamos la herencia, existen dos términos para referirse a las clases:

- La clase padre o también conocida como clase base.
- La clase hija o también conocida como clase derivada.

La clase padre o clase base, es la clase que se debe crear primero, donde se encuentra escrito el código que contiene a las variables y métodos que se van a reutilizar. Es decir, en esta clase declaramos las variables y creamos los métodos con los cuales va a funcionar parte de nuestro programa.

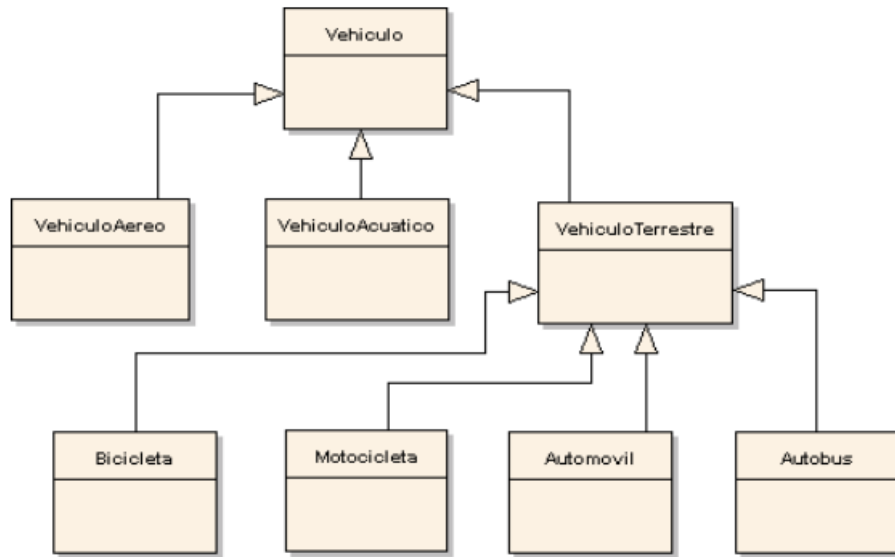
La clase hija o clase derivada, es la nueva clase, donde vamos a reutilizar los métodos y atributos, que se crearon en la clase padre sin necesidad de volver a escribir el mismo código para poder utilizarlos. Es decir, con la herencia, la clase hija puede tomar las variables y funciones que tiene su clase padre y utilizarlas sin ningún problema.

En programación, existen dos tipos de herencia:

- La herencia simple
- La herencia múltiple (No es permitido en C#)

La herencia simple consiste en que una clase hija solo puede heredar los atributos y métodos de una única clase padre. Es decir, una clase hija solo puede tener una clase padre, no puede tener más de un padre, mientras que una clase padre, su puede tener múltiples clases hijas.

Para comprender mejor la herencia y la naturaleza de las clases heredadas, es necesario entender que una clase hija es más específica conceptualmente que la clase padre, a la vez que la clase padre es más general conceptualmente que la clase hija. Esto lo podemos visualizar mejor de la siguiente manera:

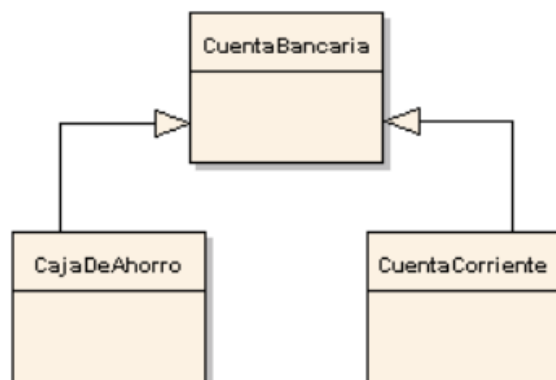


En este diagrama podemos ver a la clase padre “Vehículo” y su árbol de herencia. Por comodidad no hemos colocado las clases hijas de las clases “VehículoAereo” y “VehículoAcuatico”. ¿Qué tendrían en común las clases “Avion” y “Motocicleta”? Que ambas clases tienen como superclase a la clase vehículo, es decir, ambas SON “Vehículo”.

Es importante destacar que una clase ES UN cualquiera de sus superclases, es decir: “Motocicleta” ES UN “VehículoTerrestre” y ES UN “Vehículo”.

Con cada subclase, se amplía la cantidad de elementos de dicha clase con respecto a sus superclases, volviéndola más específica. Así, por ejemplo, la clase vehículo hace referencia a cualquier tipo de transporte, en cambio la clase “VehículoTerrestre” hace referencia a cualquier tipo de transporte que se desplace en un medio sólido, y la clase “Automóvil” hace referencia a un tipo de transporte específico. Podríamos ir más allá en el árbol de herencia y crear clases para cada modelo de automóvil y hacer que extiendan a la clase “Automóvil”.

Otro árbol de herencia podría ser el siguiente:



En este caso, la clase padre es “CuentaBancaria” y las clases hijas son “CajaDeAhorro” y “CuentaCorriente”, donde “CajaDeAhorro” ES UNA “CuentaBancaria” y “CuentaCorriente” ES UNA “CuentaBancaria”.

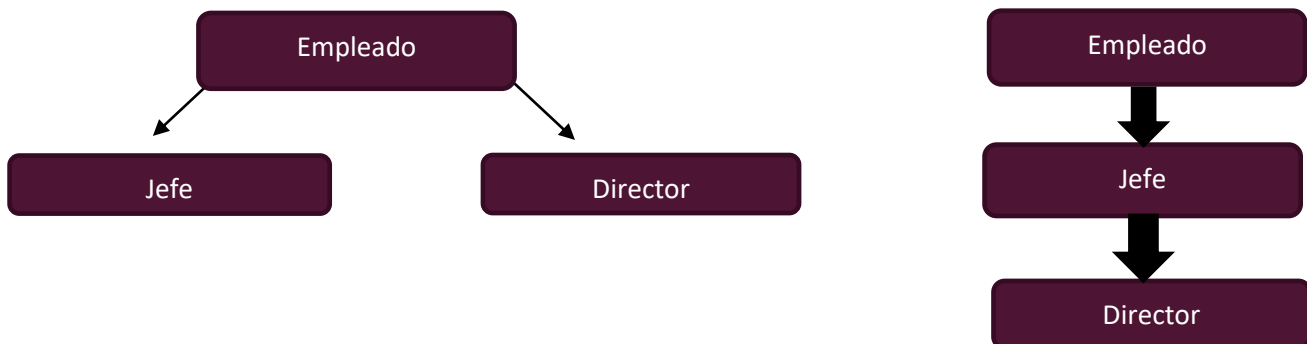
Resumiendo, el mecanismo de herencia es la capacidad que tienen las clases de recibir el comportamiento (operaciones) y estado (atributos) de otras clases.

Para definir qué clase hereda de cual hay que realizar una pregunta de ida y vuelta: “Es un?”, si la respuesta es “Si” va a heredar de dicha clase. Las clases de la zona inferior siempre son más específicas que las de la cúspide.

Por ejemplo, tenemos tres clases: Empleado, Jefe, Director. Por lo tanto tendremos que realizar las siguientes preguntas:

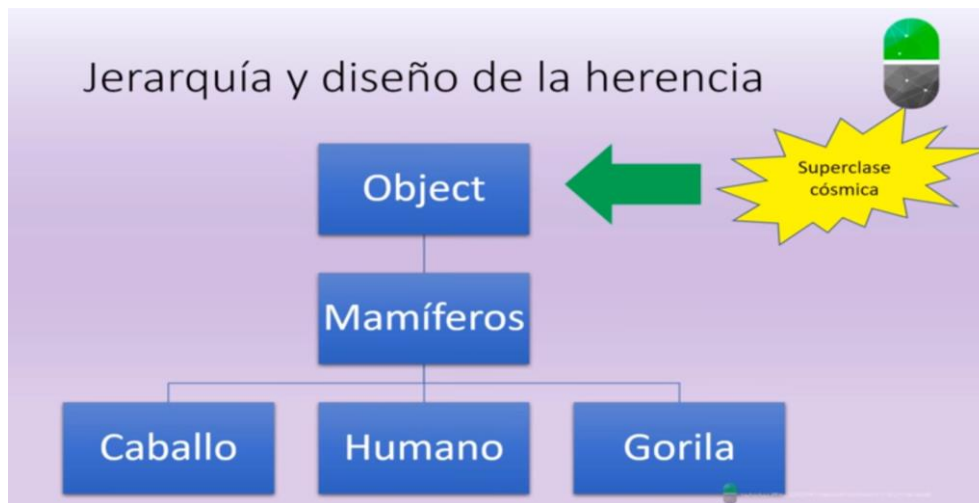
- ¿Un empleado es un jefe? No necesariamente.
- ¿Un empleado es un director? No necesariamente.
- ¿Un jefe es un empleado? SI. Por lo tanto, la clase jefe hereda de empleado
- ¿Un jefe es un director? No necesariamente.
- ¿Un director es un empleado? SI. Por lo tanto, la clase director hereda de empleado
- ¿Un director es un jefe? No necesariamente / En otro escenario podría serlo.

Y la jerarquía de clases quedaría de la siguiente manera:



Una vez armada la jerarquía de clases hay que ver las características comunes que tienen las clases. Por ejemplo: Nombre, Edad, Fecha de Alta, Salario. Y comportamientos comunes: Trabajan, Generan informes. Si son comunes se programan en la cúspide.

Otro ejemplo podría ser:



Donde mamíferos es una superclase y tanto caballo como humano y gorila son subclases. Estos tienen características comunes como: ser vertebrados, de sangre caliente, cuidado parental, amamantan. Es por eso que todos heredan de una superclase como Mamíferos.

Sintaxis: `class` Caballo : Mamiferos

Esto permite que al instanciar una clase “Caballo, no solo voy a tener los métodos de dicha clase, sino también los métodos de la clase mamíferos que es heredada por esta.

Un tema a considerar es que si realizamos un constructor que no sea el de por defecto en la clase padre Mamíferos donde recibe por parámetro un dato como puede ser el nombre del ser vivo, tendremos que aclarar en las clases hijas a que constructor nos hacemos referencia y enviarles el parámetro solicitado.

Ejemplo:

```

Class Mamiferos
{
    private string nombreSerVivo;
    public Mamiferos (string Nombre)
    {
        this.nombreSerVivo = Nombre;
    }
}

Class Humano : Mamiferos
{
    public Humano (string NombreHumano) : base (NombreHumano)
    {
    }
}
  
```


PRINCIPIO DE SUSTITUCIÓN

El principio de sustitucion ayuda a generar la jerarquia de herencia al momento de instanciar una clase. Lo haremos mediante la pregunta “¿es siempre un..?”. por ejemplo al momento de instanciar una clase preguntamos:

- ¿Un Mamifero es siempre un Caballo? No, puede ser muchos animales.
- ¿Un Caballo es siempre un Mamifero? Si, por lo tanto una variable de tipo mamifero puede contener un objeto de tipo caballo/humano/gorila.

Este principio de sustitucion se aplica al momento de utilizar el Polimorfismo.



```

class Mamifero
{
    public virtual void EmitirSonido()
    {
        Console.WriteLine("El mamífero emite un sonido.");
    }
}

class Caballo : Mamifero
{
    public override void EmitirSonido()
    {
        Console.WriteLine("El caballo relincha.");
    }
}

class Humano : Mamifero
{
    public override void EmitirSonido()
  
```

```

    {
        Console.WriteLine("El humano habla.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Mamifero mamifero1 = new Caballo();
        mamifero1.EmitirSonido(); // Salida: "El caballo relincha."

        Mamifero mamifero2 = new Humano();
        mamifero2.EmitirSonido(); // Salida: "El humano habla."
    }
}

```

INTERFACES

Las interfaces son conjuntos de directrices que deben cumplir las clases que deseen implementarlas (las clases se heredan, mientras que las interfaces se implementan, ambas se realizan de la misma manera).

En las interfaces solo estarán las definiciones de los métodos, es decir, las directrices a seguir por una clase o los comportamientos. Una interfaz va a tener, simplemente, una declaración de método/s (que van a estar sin completar). Estos métodos van a marcar los comportamientos que deben seguir aquellas clases que implementen esa interfaz.

Por convención, los nombres de las interfaces deben comenzar con la letra I mayúscula. Por ejemplo:

```
interface IPureba{}
```

Los métodos de una interfaz no se desarrollan, no tienen código, por ejemplo: `int NumPatas();` solo se declaran. Tampoco deben llevar modificadores de acceso, por defecto siempre van a ser públicos, si los declaramos privados no podremos acceder a ellos luego desde la clase donde vamos a implementar dicha interfaz.

En la clase donde implementemos la interfaz estamos OBLIGADOS a desarrollar todos los métodos de la misma. Aquí deberemos tener algunas consideraciones importantes:

- El nombre de los métodos debe coincidir con los de la interfaz.
- Deben devolver el mismo tipo de datos que los declarados en la interfaz.
- Deben tener el mismo tipo y número de parámetros que los declarados en la interfaz.

La interfaz se utiliza ya que nos genera la obligación de desarrollar los métodos de la misma, de esta manera futuros programadores no podrán obviarlos.

En caso de que una clase herede de otra y a su vez también de una interfaz, primero deberán heredar la clase y luego las interfaces. Por ejemplo: `class ave : animal, IAnimalesTerrestres`

En caso de que haya distintas interfaces implementadas con el mismo nombre de métodos es un problema llamado “ambigüedad”. Este problema se soluciona de la siguiente manera: primero se debe borrar el modificador de acceso “public” del método de la clase hija, para luego agregarle entre el tipo de datos y el nombre del método, el nombre de la interfaz al que va a pertenecer, por ejemplo: `int`

```
IAnimalesTerrestres.NumeroPatas(){}

```

RESTRICCIONES

Las interfaces presentan diversas **restricciones**:

- No se permiten definir campos (variables).
- No se pueden definir constructores.
- No se permiten definir destructores.
- No se permiten especificar modificadores de acceso en métodos (todos deben ser public por defecto).
- No se pueden anidar clases ni otro tipo de estructuras en las interfaces.

POLIMORFISMO

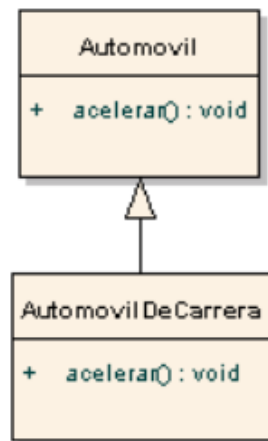
El polimorfismo en la POO, es la capacidad que se le da a un método, de comportarse de manera diferente de acuerdo a la instancia creada. Es decir, dependiendo de la clase con la que se esté interactuando, será la función que va a ejecutar el método.

Por ejemplo, supongamos que Matías es el método y recibe dos llamadas telefónicas. Matías va a comportarse de manera diferente, dependiendo de quien se esté comunicando con él. Si es una empresa la que se comunica con Matías, se comportará de manera formal y atenta. Pero si Matías recibe una llamada telefónica de un amigo, se comportará de manera amigable y divertida.

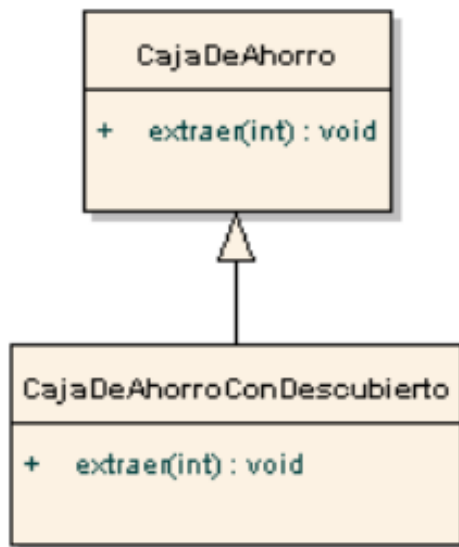
En POO encontramos dos tipos de polimorfismo, uno de ellos ya lo vimos y se conoce comúnmente como "Sobrecarga de Operaciones". El otro, el que veremos ahora, tiene que ver con la capacidad que tienen las clases de redefinir el comportamiento de una operación heredada de la clase padre. Esto significa que, en una clase padre conceptualmente más general, al tener un método al cual le dimos cierto comportamiento básico,

que puede no ser exactamente lo que debe suceder al invocar este método desde un objeto correspondiente a una clase hija, debemos sobrescribir esa operación en la clase hija con la funcionalidad o comportamiento apropiado. Si la clase hija no redefiniera ese comportamiento, el comportamiento sería el mismo de la clase padre (tras invocar a dicho método). Por lo cual, tendríamos una clase padre con la operación “hacerAlgo” que hace “x” cosa, y una clase hija con la operación “hacerAlgo” que hace “y” cosa, es decir que la misma operación tiene comportamientos distintos según la clase.

En el siguiente caso:



La clase “AutomovilDeCarrera” debe reemplazar el comportamiento de la operación “acelerar()” heredado de la clase “Automóvil”, ya que el tipo de aceleración es completamente distinta. Lo mismo sucede con las siguientes clases:



En este caso, las operaciones “extraer()” de ambas clases deben tener comportamientos distintos debido a que el límite de dinero a extraer es distinto en cada clase. La clase “CajaDeAhorro” no tiene la posibilidad de

extraer en descubierto, sin embargo, la clase “CajaDeAhorroConDescubierto” permite extraer dinero aun cuando ya no hay mas saldo en la caja. Es decir que ambas clases permiten extraer dinero, pero de formas distintas.

En conclusión, el polimorfismo consiste en hacer que un método se comuniquen con clases diferentes, y dependiendo de la clase con la que tenga comunicación, su comportamiento será completamente diferente.

En la clase padre en el método que se va a comportar diferente de acuerdo al contexto se le agrega la palabra reservada “**Virtual**”. Y en la clase hija se va a sobrescribir dicho método agregándole la palabra “**Override**”.

Ejemplo:

```
class Program
{
    static void Main(string[] args)
    {
        string valor;
        int opcion;

        Console.WriteLine("Elije una mascota: 1-Perro, 2-Ave, 3-Pez");
        valor = Console.ReadLine();
        opcion = Convert.ToInt32(valor);

        // SIN Polimorfismo
        /*if (opcion == 1)
        {
            perro miPerro = new perro();
            miPerro.Nombre = "pepe";
            miPerro.Moverse();
        }
        if (opcion == 2)
        {
            ave miAve = new ave();
            miAve.Nombre = "pio";
            miAve.Moverse();
        }
        if (opcion == 3)
        {
            pez miPez = new pez();
            miPez.Nombre = "nemo";
            miPez.Moverse();
        }

        Console.WriteLine("-----");
        */

        // CON Polimorfismo
        //creamos la variable que tendra polimorfismo
        animal miMascota = new animal();

        //basado en la selección le damos el comportamiento seleccionado
```

```

bool numpatas = false;
bool deporte = false;
perro miPerro = new perro();

// Utilizamos el principio de sustitución para poder elegir el método
// deseado de cada interfaz
IAmalesTerrestres ImiPerro = new perro();
ISaltoConPatatas ImiPerro2 = new perro();

if (opcion == 1)
{
    miMascota = new perro();
    numpatas = true;
    deporte = true;
}
if (opcion == 2)
{
    miMascota = new ave();
}
if (opcion == 3)
{
    miMascota = new pez();
}

// Ahora trabajamos en términos del concepto Animal y no en termino de clases
// especificas
Console.WriteLine("Dame el nombre");
miMascota.Nombre = Console.ReadLine();
miMascota.Moverse();

//implementacion de la interfaz
if (numpatas == true && deporte == true)
{
    Console.WriteLine("El perro tiene {0} patas", ImiPerro.NumeroPatatas());
    Console.WriteLine("El perro salta con {0} patas", miPerro2.NumeroPatatas());
    Console.WriteLine("El perro puede realizar el deporte de: " +
        miPerro.TipoDeporte());
}
}

// INTERFAZ
interface IAmalesTerrestres
{
    int NumeroPatatas();
}

interface IAmalesYDeportes
{
    string TipoDeporte();
    Boolean isOlimpico();
}

interface ISaltoConPatatas
{

```

```

        int NumeroPatas();
    }

    class animal
    {
        protected string nombre;

        public string Nombre
        {
            get
            {
                return nombre;
            }
            set
            {
                nombre = value;
            }
        }

        public virtual void Moverse()
        {
            Console.WriteLine("El animal se mueve");
        }
    }

    class ave : animal, IAnimalesTerrestres
    {
        public override void Moverse()
        {
            Console.WriteLine("El ave {0} vuela", nombre);
        }

        public int NumeroPatas()
        {
            return 2;
        }
    }

    class perro : animal, IAnimalesTerrestres, IAnimalesYDeportes,
        ISaltoConPatas
    {
        public override void Moverse()
        {
            Console.WriteLine("El perro {0} camina", nombre);
        }

        int IAnimalesTerrestres.NumeroPatas()
        {
            return 4;
        }

        int ISaltoConPatas.NumeroPatas()
        {
            return 2;
        }
    }

```

```
    }

    public string TipoDeporte()
    {
        return "Carreras";
    }

    public Boolean isOlimpico()
    {
        return false;
    }
}

class pez : animal
{
    public override void Moverse()
    {
        Console.WriteLine("El pez {0} nada", nombre);
    }
}
```

CLASES ANÓNIMAS

Las clases anónimas son clases sin nombre que se utilizan para simplificar el código en situaciones específicas, como consultas SQL o la creación de objetos temporales con propiedades definidas. Estas clases tienen ciertos requisitos y restricciones en su estructura.

En primer lugar, las clases anónimas solo pueden tener campos públicos. Esto significa que no se pueden definir propiedades con getters y setters ni otros modificadores de acceso como privados o protegidos. Los campos públicos permiten acceder directamente a los valores de las propiedades de la clase anónima.

Además, todos los campos de una clase anónima deben estar inicializados en el momento de su creación. Esto garantiza que todos los campos tendrán un valor asignado y evitarán errores al intentar acceder a ellos más adelante en el código.

Otra restricción es que los campos de una clase anónima no pueden ser estáticos. Esto significa que no se pueden definir campos que pertenezcan a la clase en sí mismo en lugar de pertenecer a las instancias individuales de la clase.

Finalmente, no se pueden definir métodos en las clases anónimas. Estas clases se utilizan principalmente para almacenar datos y no para definir comportamiento adicional.

Ejemplo:

```
var miVariable = new {Nombre= "Matias", edad= 24};  
LblNombre.Text= miVariable.Nombre;  
LblEdad.Text= miVariable.edad;
```

En este ejemplo, se crea una instancia de una clase anónima utilizando la sintaxis `new { Nombre = "Matias", Edad = 24 }`. Esta instancia tiene dos campos públicos, "Nombre" y "Edad", con sus respectivos valores asignados.

Luego, acceda a los valores de los campos de la clase anónima utilizando la notación de punto. En este caso, asignamos el valor del campo "Nombre" a un control de texto llamado "LblNombre" y el valor del campo "Edad" a otro control de texto llamado "LblEdad".

Las clases anónimas proporcionan una forma rápida y conveniente de crear objetos temporales con propiedades predefinidas sin necesidad de definir una clase separada. Sin embargo, debido a sus restricciones y limitaciones, se recomienda utilizarlas solo en situaciones específicas donde su simplicidad y concisión sean beneficiosas.

CLASES ABSTRACTAS

Una clase abstracta es una clase que no se puede instanciar directamente, es decir, no se pueden crear objetos de esa clase. En cambio, se utiliza como una clase base para otras clases que la heredan. Las clases abstractas pueden contener implementaciones de métodos, propiedades, eventos, campos y otros miembros, al igual que las clases regulares. Sin embargo, también pueden tener miembros abstractos, que son declaraciones de métodos (sin implementación) que deben ser implementados en las clases derivadas.

Para declarar una clase abstracta, se utiliza la palabra clave "abstract" antes de la declaración de la clase. Los miembros abstractos se declaran utilizando la palabra clave "abstract" en su firma, y las clases derivadas deben proporcionar una implementación para esos miembros.

La principal diferencia entre una clase abstracta y una interfaz es que una clase abstracta puede tener implementaciones concretas de métodos y propiedades, mientras que una interfaz solo puede contener la firma de los miembros, sin ninguna implementación.

Las clases abstractas se utilizan cuando se desea proporcionar una implementación base común para las clases derivadas y se espera que esas clases derivadas implementen o modifiquen algunos de los miembros. Proporcionan una estructura base para compartir código y comportamiento común entre varias clases relacionadas.

Por otro lado, las interfaces se utilizan para definir un conjunto de métodos y propiedades que deben ser implementadas por cualquier clase que las utilicen. No pueden contener implementaciones concretas de métodos. Las interfaces permiten establecer contratos comunes entre clases no relacionadas, lo que permite la abstracción y el polimorfismo.

Las clases concretas son aquellas que heredan de la clase abstracta y garantizan una implementación completa de todos los métodos y funcionalidades definidas en la clase abstracta. Estas clases concretas son las que realmente pueden ser instanciadas y utilizadas en el código.

Un método abstracto es un método declarado en la clase abstracta pero sin proporcionar una implementación concreta en esa clase. Su implementación se deja para las clases concretas que heredan de la clase abstracta. El propósito de los métodos abstractos es definir una interfaz común que todas las clases concretas deben implementar.

Al declarar una clase como abstracta, se evita que los usuarios finales de la clase puedan crear objetos directamente de ella. Es importante tener en cuenta que, si una clase contiene al menos un método abstracto, esa clase debe ser declarada como abstracta.

MÉTODO ABSTRACTO

Un método abstracto es un tipo de método en una clase (o interfaz) que se declara sin proporcionar una implementación concreta. Es decir, solo se declara su firma o estructura, pero no se define el código que se ejecutará en el cuerpo del método.

El propósito principal de un método abstracto es servir como un "punto de entrada" para que las subclases que hereden y proporcionen su propia implementación. Un método abstracto actúa como un contrato que establece que todas las subclases deben implementar ese método y proporcionar su propia lógica de ejecución.

CLASES SELLADAS

Una clase sellada, también conocida como clase final o seal class en algunos lenguajes de programación, es una clase que no puede ser heredada por otras clases. Es decir, una vez que una clase se declara como sellada, no se puede derivar o extender de ella para crear subclases adicionales.

La intención detrás de una clase sellada es indicar que la clase está modificada para ser final y no debe ser modificada o extendida. Se utiliza cuando se desea guardar la herencia y evitar que se agreguen nuevas funcionalidades o se modifiquen las características existentes de la clase en subclases.

Al declarar una clase como sellada, se impide que otras clases hereden de ella utilizando la palabra clave "sealed" o "final" dependiendo del lenguaje de programación. Esto se realiza al final de la declaración de la clase. Por ejemplo, en C#, se utiliza la palabra clave "sealed":

```
sealed class MiClaseSellada
{
    // Contenido de la clase
}
```

Del mismo modo, los métodos también se pueden marcar como sellados para evitar que se sobrescriban en las subclases. Esto se logra utilizando la palabra clave "sealed" o "final" antes de la declaración del método que se desea sellar. Por ejemplo, en C#:

```
class MiClaseBase
{
    public virtual void MetodoVirtual()
    {
        // Código del método
    }
}
class MiClaseDerivada : MiClaseBase
{
    public sealed override void MetodoVirtual()
    {
        // Código del método
    }
}
```

En este ejemplo, la clase `MiClaseDerivada` hereda de `MiClaseBase` sobrescribe el método virtual `MetodoVirtual()`. Al utilizar la palabra clave "sealed" en la clase derivada, se impide que cualquier clase posterior pueda sobrescribir nuevamente el método.

La utilización de clases selladas y métodos sellados puede ser útil cuando se desea tener un control estricto sobre la jerarquía de herencia y evitar cambios no deseados o incompatibles en el comportamiento de las clases o métodos base.

STRUCTS (ESTRUCTURAS)

La diferencia clave entre las estructuras y las clases radica en su naturaleza y comportamiento. Aquí hay algunas características distintivas de cada uno:

- ❖ **Semántica de valor vs. semántica de referencia:** Las estructuras son tipos de valor, lo que significa que cuando se asignan o se pasan como argumentos, se copia el valor completo. Por otro lado, las clases son tipos de referencia, lo que implica que cuando se asignan o se pasan como argumentos, se comparte la referencia al objeto en la memoria.
- ❖ **Almacenamiento en la pila vs. almacenamiento en el heap:** Tanto las estructuras como las clases se almacenan en el heap. Sin embargo, cuando una estructura se declara dentro de un método, se almacena en la pila en lugar de en el montón. Esto se debe a que las estructuras son tipos de valor y su duración está asociada con la duración del método en el que se declaran.
- ❖ **Herencia:** Las clases documentan la herencia, lo que significa que una clase puede heredar propiedades y comportamientos de otra clase. Las estructuras no pueden heredar de otras estructuras ni de clases.
- ❖ **Comportamiento predeterminado:** Las estructuras tienen un constructor predeterminado que inicializa automáticamente todos sus miembros a sus valores predeterminados. Las clases no tienen un constructor predeterminado y deben ser inicializadas limpiamente.

Las estructuras (structs) se utilizan en situaciones en las que se necesita representar un conjunto de datos relacionados como una unidad indivisible. A continuación, se presentan algunos casos comunes en los que las estructuras son útiles:

- ❖ **Tipos de datos pequeños y livianos:** Las estructuras son adecuadas para representar tipos de datos pequeños y livianos, como coordenadas (x, y), puntos en un espacio tridimensional, colores, fechas, intervalos de tiempo, etc. Estos tipos de datos no requieren referencias o comportamientos complejos y se benefician de la semántica de valor de las estructuras.
- ❖ **Pasar datos por valor:** Cuando se necesita pasar datos por valor en lugar de por referencia, las estructuras son útiles. Al pasar una estructura como argumento a un método o al aprobarla a otra variable, se realiza una copia completa del valor de la estructura. Esto puede ser útil en situaciones en las que se desea evitar cambios inesperados en los datos originales.
- ❖ **Estructuras de datos simples:** Las estructuras se pueden utilizar para crear estructuras de datos simples, como pilas, colas, listas enlazadas, diccionarios, etc.
- ❖ **Rendimiento:** Debido a su naturaleza de valor ya su almacenamiento en la pila en algunos contextos, las estructuras pueden ser más eficientes en términos de rendimiento que las clases. Esto se debe a que no requiere requerir memoria en el montón ya que su acceso es más rápido.

Es importante tener en cuenta que el uso de estructuras o clases depende del contexto y los requisitos específicos de su aplicación. Si necesita realizar operaciones complejas, necesita herencia u otros conceptos de orientación a objetos, o si los datos que maneja son grandes y complejos, es posible que las clases sean

más adecuadas. En general, las estructuras se utilizan para representar tipos de datos simples y livianos que se benefician de la semántica de valor y el rendimiento optimizado.

ENUM (TIPOS ENUMERADOS)

Los tipos enumerados son un conjunto de constantes con nombre que sirven para representar y manejar valores fijos (constantes) en un programa. Por ejemplo las estaciones del año, tenemos un conjunto llamado "Estaciones" que dentro contienen: Verano, Otoño, Invierno y Primavera (`enum Estaciones{ Verano, Otoño, Invierno, Primavera }`).

Los tipos enumerados (enum) son una forma de definir un conjunto de constantes con nombre en un programa. Se utilizan para representar y manejar valores fijos o constantes que pertenecen a un dominio específico.

En el caso de las estaciones del año, podemos definir un tipo enumerado llamado "Estaciones" que contiene las constantes: Verano, Otoño, Invierno y Primavera. La sintaxis en C# para definir un tipo enumerado sería similar a esto: `enum Estaciones{ Verano, Otoño, Invierno, Primavera }`

Las constantes dentro del enum se nombran en mayúsculas y pueden ser accedidas utilizando el nombre del tipo enumerado seguido del nombre de la constante, por ejemplo: `Estaciones.Verano`, `Estaciones.Otoño`, etc.

Los tipos enumerados tuvieron varias **ventajas**, como:

- ❖ **Legibilidad del código:** Al utilizar un tipo enumerado, se hace más legible y comprensible el código, ya que se utilizan nombres descriptivos en lugar de valores numéricos o literales.
- ❖ **Seguridad en tiempo de compilación:** Al usar un tipo enumerado, el compilador verifica que los valores asignados sean válidos y pertenezcan al conjunto de constantes definidas en el enum. Esto ayuda a evitar errores de eliminación de valores incorrectos.
- ❖ **Intellisense y autocompletado:** Los IDE y editores de código suelen proporcionar sugerencias automáticas y completar los nombres de las constantes enumeradas, lo que facilita la escritura del código y evita errores de escritura.

Ejemplo:

```
Enum Bonus {Bajo = 500, Normal = 1000, Bueno = 1500, Extra = 3000}  
class empleado  
{  
    private double bonus, salario;  
    public empleado (Bonus bonusEmpleado, double Salario)  
    {
```

```

        bonus = (double) bonusEmpleado;
        This.salario = Salario;
    }
    public double getSalario
    {
        Return salario + bonus;
    }
}

```

PROGRAMACIÓN GENÉRICA

La programación genérica es una técnica que permite crear clases y métodos que pueden trabajar con diferentes tipos de datos de manera segura y eficiente. En lugar de definir clases específicas para cada tipo de dato que se desea manejar, se utilizan clases genéricas que actúan como "comodines" y pueden adaptarse a diferentes tipos de objetos.

La principal ventaja de la programación genérica es la reutilización de código, ya que una única clase genérica puede manejar diferentes tipos de objetos sin necesidad de crear múltiples versiones de la misma. Esto simplifica el código y evita la duplicación necesaria.

Ejemplo:

```

// Clase genérica que maneja una colección de elementos de tipo T
class Coleccion<T>
{
    private List<T> elementos;

    public Coleccion()
    {
        elementos = new List<T>();
    }

    public void Agregar(T elemento)
    {
        elementos.Add(elemento);
    }

    public T Obtener(int indice)
    {
        return elementos[indice];
    }
}

internal class Program
{
    // Uso de la clase genérica
    static void Main()
    {
        // Creación de una instancia de Coleccion que maneja enteros
        Coleccion<int> numeros = new Coleccion<int>();
        numeros.Agregar(1);
        numeros.Agregar(2);
        int resultado = numeros.Obtener(0); // Obtiene el primer elemento
    }
}

```

```
        // Creación de una instancia de Coleccion que maneja cadenas de texto
        Coleccion<string> palabras = new Coleccion<string>();
        palabras.Agregar("Hola");
        palabras.Agregar("Mundo");
        string texto = palabras.Obtener(1); // Obtiene el segundo elemento
    }
}
```

En el ejemplo anterior, se define la clase genérica `Coleccion<T>` que puede manejar una colección de cualquier tipo de elemento. Se utiliza el parámetro de tipo `T` para indicar el tipo de dato que se manejará. En el método `Agregar`, se puede agregar un elemento de tipo `T` a la colección, y en el método `Obtener`, se puede obtener un elemento de tipo `T` en base a un índice.

Al utilizar la programación genérica, evitamos el uso continuo del casting o casteo de tipos, ya que la clase genérica se adapta automáticamente al tipo de dato que se está manejando. Además, los errores se detectan en el tiempo de compilación, lo que brinda una mayor seguridad y facilita la corrección de problemas.