

# Fundamentos de la programación orientada a objetos

## Contenido

Descripción general	1
Clases y objetos	2
Uso de la encapsulación	10
El lenguaje C# y la orientación a objetos	23
Definición de sistemas orientados a objetos	36

## Notas para el instructor

Este módulo proporciona a los estudiantes la teoría, los conceptos y la terminología básica de la programación orientada a objetos. También incluye una porción mínima de sintaxis de C#, concretamente la necesaria para la encapsulación.

Al final de este módulo, los estudiantes serán capaces de:

- Definir los términos *objeto* y *clase* en el contexto de la programación orientada a objetos.
- Definir los tres aspectos básicos de un objeto: identidad, estado y comportamiento.
- Describir la abstracción y cómo ayuda a crear clases reutilizables que son fáciles de mantener.
- Usar la encapsulación para combinar métodos y datos en una sola clase y forzar la abstracción.
- Explicar los conceptos de herencia y polimorfismo.
- Crear y utilizar clases en C#.



## ◆ Clases y objetos

**Objetivo del tema**

Ofrecer una introducción a los temas tratados en esta sección.

**Explicación previa**

Los términos *clase* y *objeto* aparecen muy a menudo. En esta sección aprenderemos qué es lo que significan exactamente.

- ¿Qué es una clase?
- ¿Qué es un objeto?
- Comparación de clases y estructuras
- Abstracción

---

Toda la estructura de C# está basada en el modelo de programación orientada a objetos. Para sacar el máximo partido a C# como lenguaje es necesario comprender la naturaleza de la programación orientada a objetos.

Al final de esta lección, usted será capaz de:

- Definir los términos *objeto* y *clase* en el contexto de la programación orientada a objetos.
- Aplicar el concepto de abstracción.

## ¿Qué es una clase?

**Objetivo del tema**

Explicar el concepto de clase.

**Explicación previa**

El objetivo principal del lenguaje C# es definir clases y especificar su comportamiento.

**■ Para el filósofo...**

- Un artefacto de *clasificación* humana
- *Clasificamos* según un comportamiento o atributos comunes
- Acordamos descripciones y nombres de *clases* útiles
- Creamos vocabulario; nos comunicamos; ¡pensamos!

**■ Para el programador orientado a objetos...**

- Una construcción sintáctica con nombre que describe un comportamiento y atributos comunes
- Una estructura de datos que incluye datos y funciones

La palabra *clase* proviene de clasificación. Formar clases es el acto de clasificar, algo que hacen todos los seres humanos (y no sólo los programadores). Por ejemplo, todos los coches comparten un mismo comportamiento (se pueden dirigir, detener,...) y atributos comunes (tienen cuatro ruedas, un motor,...). Usamos la palabra *coche* para referirnos a todos estos comportamientos y propiedades comunes. Imaginemos qué ocurriría si no fuésemos capaces de clasificarlos en un concepto al que damos un nombre. En lugar de *coche*, tendríamos que decir todas las cosas que *coche* significa. Las frases se harían largas y complicadas, y de hecho es muy probable que la comunicación fuera imposible. En la medida en que todos estemos de acuerdo en lo que significa una palabra (es decir, en la medida en que todos hablemos el mismo idioma), podremos comunicarnos y expresar ideas complejas, pero precisas, de una forma compacta. Estos conceptos con nombre nos sirven a su vez para crear conceptos de un nivel más alto y para aumentar la capacidad expresiva de la comunicación.

Todos los lenguajes de programación describen datos y funciones comunes. Esta capacidad de describir características comunes ayuda a evitar la duplicación. Uno de los lemas básicos de la programación es “No te repitas”. Un código duplicado crea problemas porque es más difícil de mantener. Por el contrario, un código que no se repite es más fácil de mantener, es parte porque es más corto. Los lenguajes orientados a objetos llevan este concepto al siguiente nivel, ya que permiten descripciones de clases (conjuntos de objetos) que comparten estructura y comportamiento. Si se hace bien, este principio funciona extremadamente bien y se adapta de forma natural a la forma de hablar y comunicarse que tienen las personas.

Las clases no se limitan a clasificar objetos concretos (como coches), sino que también se pueden usar para clasificar objetos abstractos (como tiempo). No obstante, a la hora de clasificar objetos abstractos los límites se hacen más imprecisos y el diseño se convierte en algo importante.

El único requisito real que debe cumplir una clase es estimular la comunicación.

## ¿Qué es un objeto?

**Objetivo del tema**

Definir el término *objeto*.

**Explicación previa**

A menudo se comete el error de usar los términos *clase* y *objeto* como si significaran lo mismo, cuando en realidad son dos conceptos muy diferentes.

- Un objeto es una instancia de una clase
- Los objetos se caracterizan por:
  - Identidad: Los objetos se distinguen unos de otros
  - Comportamiento: Los objetos pueden realizar tareas
  - Estado: Los objetos contienen información



**Recomendación al profesor**

Puede emplear la analogía del coche a lo largo de toda la explicación de identidad, comportamiento y estado. Por ejemplo, puede preguntar a los estudiantes si esta frase se refiere a los coches como objetos o como una clase: "El carril de lentos, que es el derecho, es para coches con más de tres ocupantes." Esta frase usa "coche" como un objeto (una instancia de una clase). Los objetos pueden ser anónimos, pero no por ello dejan de ser objetos. El hecho de que un coche lleve a tres personas no lo convierte en un objeto distinto de un coche que lleve a dos o cuatro personas. El número de ocupantes es un ejemplo del estado del coche. Explique que *estado* se refiere a los valores de los atributos internos de un objeto que pueden variar con el tiempo, como el número de pasajeros. Compare esto con los valores que probablemente estén fijos y no cambien desde la fabricación del coche, como el número de puertas.

La palabra *coche* significa distintas cosas según el contexto. A veces usamos la palabra coche para referirnos al concepto general de coche: hablamos de coche como una *clase*, la del conjunto de todos los coches, sin pensar en ningún tipo de coche concreto. En otras ocasiones empleamos la palabra coche para hablar de un coche en particular. Los programadores usan el término *objeto* o *instancia* para referirse a un coche concreto. Es importante entender esta diferencia.

Los objetos se pueden comprender mejor con las tres características de identidad, comportamiento y estado.

## Identidad

La identidad es la característica que distingue un objeto de todos los demás objetos de la misma clase. Por ejemplo, imaginemos que dos vecinos tienen coches que son exactamente de la misma marca, modelo y color. A pesar de lo mucho que se puedan parecer, podemos estar seguros de que sus matrículas serán diferentes y reflejarán la identidad de cada uno de los coches. La ley dice que es necesario poder distinguir un objeto coche de otro. (¿Cómo les iría a los seguros de automóviles si los coches no tuvieran identidad?)



## Comportamiento

El comportamiento es la característica que hace que los objetos sean útiles. Los objetos existen para que se comporten de una cierta manera. La mayor parte del tiempo podemos olvidarnos de cómo funciona un coche y pensar sólo en su comportamiento externo o de alto nivel. Los coches nos son útiles porque podemos conducirlos; tienen un mecanismo interno, pero nos resulta casi inaccesible. Lo que sí es accesible es el comportamiento del objeto, que es además el factor determinante para su clasificación. Los objetos de una misma clase comparten un comportamiento común: Un coche es un coche porque podemos conducirlo, un bolígrafo es un bolígrafo porque podemos usarlo para escribir.

## Estado

El *estado* se refiere a los mecanismos internos de un objeto que hacen que se comporte de una determinada manera. Un objeto bien diseñado mantiene su estado inaccesible. Esto guarda una relación muy estrecha con los conceptos de abstracción y encapsulación. No nos preocupa por qué un objeto hace lo que hace, sólo queremos que lo haga. Es posible que dos objetos tengan el mismo estado, pero siguen siendo dos objetos diferentes; por ejemplo, dos gemelos idénticos tienen exactamente el mismo estado (su ADN), pero son dos personas distintas.

## Comparación de clases y estructuras

### Objetivo del tema

Ofrecer una breve comparación entre clases y estructuras.

### Explicación previa

Tal vez sepa ya que las estructuras pueden contener métodos y datos, igual que las clases. ¿Cuál es entonces la diferencia entre una estructura y una clase?

#### ■ Una estructura define un valor

- Sin identidad, estado accesible, sin comportamiento añadido

#### ■ Una clase define un objeto

- Identidad, estado inaccesible, comportamiento añadido

<pre>struct Time {     public int hour;     public int minute; }</pre>	<pre>class BankAccount {     ...     ... }</pre>
--	--

### Recomendación al profesor

En programas C++, las palabras reservadas **struct** y **class** se pueden usar indistintamente. Haga hincapié en que, en C#, las clases no son un tipo de "super estructuras."

## Estructuras

Una estructura, como *Reloj* en el ejemplo, no tiene identidad. Si dos variables *Reloj* representan las 12:30, el programa se comportará exactamente de la misma manera independientemente de cuál de ellas utilizemos. Las entidades de software sin identidad se llaman *valores*. Los tipos predefinidos descritos en el Módulo 3, "Uso de variables de tipo valor", en el Curso 2124C, *Programación en C#*, como **int**, **bool**, **decimal** y todos los tipos **struct**, se llaman en C# *tipos de valor*.

Las variables del tipo struct pueden contener métodos, aunque es preferible que no lo hagan. Idealmente, deben contener sólo datos. Por otra parte, no hay ningún problema en definir operadores en estructuras. Los operadores son un tipo de métodos que no añaden nada al comportamiento, sino sólo una sintaxis más concisa para un mismo comportamiento.

## Clases

Una clase, como **CuentaBancaria** en el ejemplo, tiene identidad. Si hay dos objetos **CuentaBancaria**, el programa se comportará de distinta forma dependiendo de cuál se utilice. Las entidades de software con identidad se llaman *objetos* (en ocasiones, las variables del tipo struct reciben también el nombre de objetos, pero estrictamente hablando son *valores*). Los tipos representados por clases se llaman en C# *tipos de referencia*. A diferencia de las estructuras, en una clase bien diseñada no se debe ver nada que no sea un método. Estos métodos añaden comportamiento de alto nivel al comportamiento primitivo que existe a nivel de los datos inaccesibles.

## Tipos de valor y tipos de referencia

Los tipos de valor son los que se encuentran en el nivel más bajo de un programa. Son los elementos que se utilizan para crear entidades de software más grandes. Los tipos de valor se pueden copiar libremente y están en la pila como variables locales o como atributos dentro de los objetos que describen.

Los tipos de referencia son los que se encuentran en el nivel más alto de un programa. Se crean a partir de entidades de software más pequeñas. Normalmente los tipos de referencia no se pueden copiar y están en el montón (heap).

## Abstracción

**Objetivo del tema**

Definir la abstracción.

**Explicación previa**

¿Es necesario saber cómo funciona una cosa para poder usarla?

**■ La abstracción es ignorancia selectiva**

- Decidir qué es importante y qué no lo es
- Concentrarse en lo importante y depender de ello
- Ignorar lo que no es importante y no depender de ello
- Usar encapsulación para forzar una abstracción

El objetivo de la abstracción es no perderse en vaguedades y crear un nuevo nivel semántico en el que se pueda ser absolutamente preciso.

Edsger Dijkstra

**Recomendación al profesor**

Los contenidos de la transparencia tienen implicaciones importantes. Lo primero que se dice es que hay que decidir qué es importante y qué no lo es. En otras palabras, hay que tomar decisiones de diseño. Dado el nivel de este curso, dedique poco o ningún tiempo a discutir este tema. Concéntrese más bien en la dependencia, que guarda una relación muy estrecha con la idea de cambio (que se discute a continuación).

La *abstracción* es la táctica de despojar una idea u objeto de todo lo innecesario hasta llegar a su forma mínima y esencial. Una buena abstracción elimina los detalles poco importantes y permite concentrarse sólo en lo que es importante.

La abstracción es un importante principio del software. Una clase bien diseñada deja ver una cantidad mínima de métodos que permiten obtener fácilmente el comportamiento esencial de la clase. Desgraciadamente, crear buenas abstracciones de software no es sencillo. Para llegar a una buena abstracción suele ser necesario tener un profundo conocimiento del problema y de su contexto, pensar con gran claridad y tener una enorme experiencia.

### Dependencia mínima

Las mejores abstracciones de software hacen que las cosas más complejas parezcan sencillas. Para ello ocultan todos los aspectos de una clase que no son esenciales. Una vez ocultos, estos aspectos ya no pueden ser vistos ni utilizados y no se depende de ellos de ninguna manera.

Este principio de dependencia mínima es lo que hace que la abstracción sea tan importante. Una de las pocas cosas seguras en el desarrollo de software es que siempre es necesario cambiar el código. La comprensión perfecta sólo se alcanza (si es que se consigue) al final del proceso de desarrollo; las primeras decisiones se toman a partir de una comprensión incompleta del problema, por lo que siempre hay que reconsiderarlas más tarde. Las especificaciones también cambian a medida que se va entendiendo mejor el problema. Las versiones posteriores incluyen cada vez más funciones. Los cambios son normales en el desarrollo de software; lo más que podemos hacer cuando se producen es minimizar sus efectos, y cuanto menos dependamos de una cosa, menos nos afectarán sus cambios.

## Citas ilustrativas

He aquí algunas citas para ilustrar el principio de dependencia mínima que hace que la abstracción sea tan importante:

### Para su información

La cita de Dijkstra se puede usar para intentar desmontar un mito muy extendido; no hay nada vago en una abstracción de software.

Tanto usted como sus estudiantes pueden encontrar más información sobre el profesor Edsger Wybe Dijkstra en [www.cs.utexas.edu/users/UTCS/report/1994/profiles/dijkstra.html](http://www.cs.utexas.edu/users/UTCS/report/1994/profiles/dijkstra.html)

Entre sus contribuciones a la informática y las matemáticas están los semáforos informáticos; sus famosos nombres "P y V" para los semáforos proceden de las palabras holandesas *passeer* y *verlaat*, o *pasar* y *salir*. También se dio su nombre al algoritmo de Dijkstra, que encuentra la ruta más corta desde un punto de un gráfico hasta su destino.

Cuanto más perfecta es una máquina, más invisible resulta tras su función. Es como si la perfección se alcanzara no cuando ya no hay nada más que añadir, sino cuando ya no queda nada que eliminar. En la cima de su evolución, la máquina queda totalmente oculta.

—Antoine de Saint-Exupéry, *Viento, arena y estrellas*

Podríamos definir el *minimum* como la perfección que alcanza un objeto cuando ya resulta imposible mejorarlo por substracción. Ésta es la calidad que define a un objeto cuando todos y cada uno de sus componentes, detalles y conexiones han sido reducidos o condensados a lo esencial. Es el resultado de la omisión de lo innecesario.

—John Pawson, *Minimum*

El objetivo básico de la comunicación es la claridad y la simplicidad. La simplicidad es el resultado de un esfuerzo bien enfocado.

—Edward de Bono, *Simplicidad*

## ◆ Uso de la encapsulación

**Objetivo del tema**

Ofrecer una introducción a los temas tratados en esta sección.

**Explicación previa**

¿Qué es la encapsulación?  
¿Cómo se puede emplear la encapsulación en una clase, y por qué?

- Combinación de datos y métodos
- Control de la visibilidad de acceso
- ¿Por qué se encapsula?
- Datos de objetos
- Uso de datos estáticos
- Uso de métodos estáticos

---

Al final de esta lección, usted será capaz de:

- Combinar datos y métodos en una sola cápsula.
- Usar encapsulación dentro de una clase.
- Usar métodos de datos estáticos en una clase.

## Combinación de datos y métodos

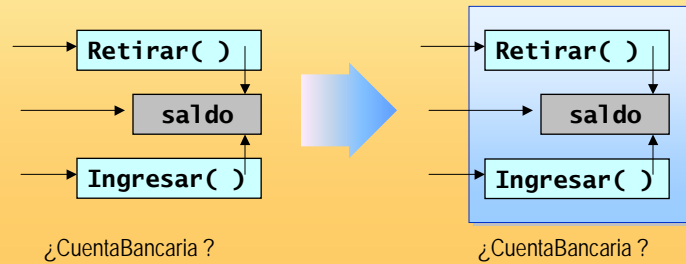
### Objetivo del tema

Discutir la motivación que hay detrás de la encapsulación.

### Explicación previa

Después de decidir qué partes pueden ser de acceso público y cuáles deben quedar ocultas, es preciso contar con una forma sencilla de ocultar la complejidad interna de una clase.

- Combinación de datos y métodos en una sola *cápsula*
- La frontera de la cápsula crea un espacio interior y otro exterior



La encapsulación incluye dos aspectos importantes:

- Combinación de datos y funciones en una sola entidad (esta transparencia)
- Control de la accesibilidad a los miembros de la entidad (siguiente transparencia)

### Recomendación al profesor

La transparencia termina recomendando el uso de la encapsulación para forzar la abstracción.

Ésta es la primera de dos transparencias sobre encapsulación. La segunda transparencia proporciona información adicional. En esta transparencia se hace hincapié en la palabra *cápsula* porque de ella se deriva la palabra *encapsulación*.

Observe que la figura de la izquierda lleva un signo de interrogación. Esto se debe a que no existe ningún elemento individual e independiente que represente la cuenta bancaria.

## Programación procedural

Los programas procedurales tradicionales, escritos en lenguajes como C, contienen fundamentalmente una gran cantidad de datos y muchas funciones, y cada función puede acceder a todos los datos. Este método con un nivel tan alto de asociación puede funcionar bien en un programa pequeño, pero se hace cada vez menos conveniente a medida que crece el programa. Un cambio en la representación de los datos puede resultar caótico, ya que hará que fallen todas las funciones que usan los datos modificados (y que por tanto dependen de ellos). Los cambios son más complicados a medida que el programa se hace más grande y, por tanto, más frágil y menos estable. La separación de datos y funciones no se mantiene al cambiar la escala del programa. Esto dificulta los cambios y, como saben todos los desarrolladores de software, el cambio es la única constante.

La separación de datos y funciones plantea otro serio problema. En términos de abstracción de comportamientos a alto nivel, esta técnica no corresponde a la forma de pensar de los humanos. Puesto que los programas están escritos por personas, es mucho mejor utilizar un modelo de programación que se aproxime a la forma en que las personas piensan y no a la forma en que están fabricados los sistemas informáticos.

## Programación orientada a objetos

La programación orientada a objetos surgió para resolver estos problemas en la medida de lo posible. Si se entiende y se utiliza debidamente, la programación orientada a objetos puede llegar a ser realmente una programación orientada a personas, puesto que los humanos tendemos de forma natural a pensar y trabajar en términos del comportamiento de objetos a alto nivel.

El paso primero y más importante para pasar de la programación procedural a la programación orientada a objetos consiste en combinar los datos y las funciones en una sola entidad.



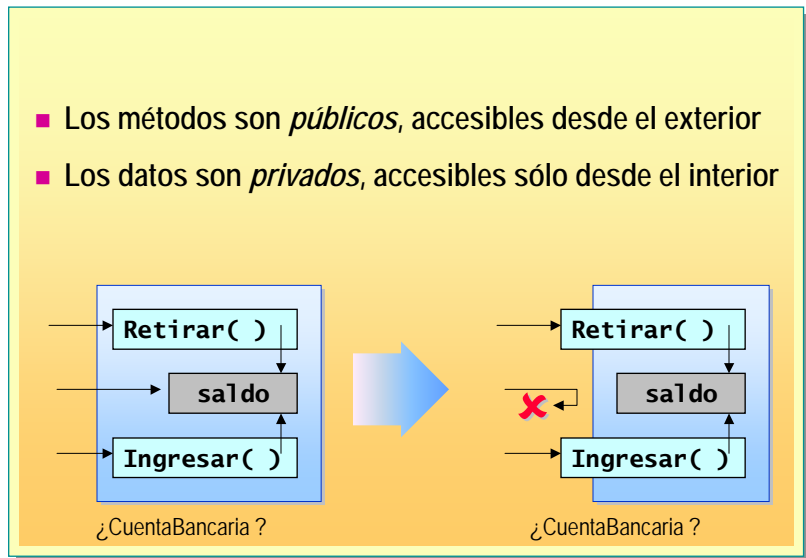
## Control de la visibilidad de acceso

### Objetivo del tema

Explicar cómo se puede llevar a la práctica la encapsulación.

### Explicación previa

Después de decidir qué partes van a quedar ocultas del mundo exterior, ¿cómo podemos asegurarnos de que nadie podrá ver cómo funcionan las clases?



### Recomendación al profesor

Esta transparencia explica el segundo aspecto de la encapsulación.

Observe que la figura de la izquierda lleva un signo de interrogación. Esto se debe a que en la vida real uno no tiene acceso directo a su saldo bancario. La figura de la derecha modifica la frontera de forma que el saldo queda en la parte interior. Como esto corresponde a las cuentas bancarias reales, no se ha utilizado signo de interrogación.

En la figura de la izquierda, **Retirar**, **Ingresar** y **saldo** han sido agrupadas dentro de una “cápsula.” La transparencia sugiere que el nombre de la cápsula es **CuentaBancaria**. Sin embargo, hay algo que no funciona en este modelo de cuenta bancaria: el dato **saldo** es accesible (si en la realidad se pudiera acceder directamente al saldo de una cuenta bancaria, sería posible aumentarlo sin hacer ningún ingreso). Las cuentas bancarias no funcionan de este modo, lo que quiere decir que hay una mala correspondencia entre el problema y su modelo.

La encapsulación permite resolver este problema. Una vez se han combinado datos y funciones en una sola entidad, ésta forma una frontera cerrada que crea de forma natural un espacio interior y otro exterior. Esta frontera se puede utilizar para controlar de forma selectiva la accesibilidad de las entidades: algunas serán accesibles desde el interior y el exterior, mientras que a otras sólo será posible acceder desde el interior. Los miembros que son siempre accesibles se denominan *públicos*, mientras que aquellos a los que sólo se puede acceder desde el interior son *privados*.

Para que el modelo de una cuenta bancaria se aproxime más a la realidad, se puede hacer que **saldo** sea privado y que los métodos **Retirar** e **Ingresar** sean públicos. De esta forma, la única manera de aumentar el saldo desde el exterior es ingresar dinero en la cuenta. Como se ve, **Ingresar** está en el interior y por tanto puede acceder a **saldo**.

Como muchos otros lenguajes de programación orientados a objetos, C# da completa libertad a la hora de elegir qué miembros deben ser accesibles. Es posible crear datos públicos siempre que se desee. No obstante, se recomienda marcar los datos siempre como privados (esta recomendación se convierte en obligatoria en algunos lenguajes de programación).

Los tipos cuya representación de datos es completamente privada se llaman tipos de datos abstractos (ADT). Son abstractos en el sentido de que no es posible acceder a la representación de datos privada (ni depender de ella); sólo se pueden usar los métodos de comportamiento.

En cierto modo, los tipos predefinidos como **int** también son ADT. Para sumar dos variables enteras no es necesario conocer la representación binaria interna de cada valor entero; tan sólo hay que saber el nombre del método que hace sumas: el operador de suma (+).

Cuando se forman miembros accesibles (públicos), es posible crear distintas vistas de una misma entidad. La vista desde el exterior es un subconjunto de la vista desde el interior. Una vista restringida está muy próxima a la idea de abstracción: desnudar la idea para quedarse con lo esencial.

La decisión de si algo debe quedar dentro o fuera implica una gran cantidad de diseño. Cuantas más características queden en el interior (y se puedan seguir utilizando), tanto mejor.

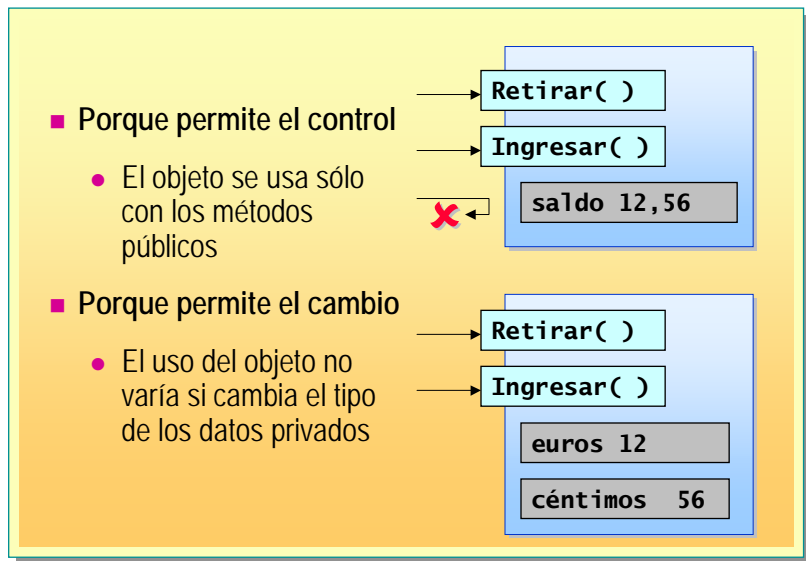
## ¿Por qué se encapsula?

### Objetivo del tema

Explicar cómo se puede usar la encapsulación para que las aplicaciones sean más fáciles de modificar y mantener.

### Explicación previa

¿Con qué frecuencia cambian los detalles internos de una clase? ¿Es posible cambiar la forma de almacenamiento o procesamiento de la información?



### Recomendación al profesor

Esta transparencia presenta el concepto de cambio y los principios fundamentales de la programación orientada a objetos. No es posible evitar los cambios; lo más que se puede hacer es trabajar con un lenguaje y un proceso de diseño que minimice sus efectos.

Observe que el saldo es de 12,56 € en los dos ejemplos de la transparencia (en distintas representaciones).

Hay dos razones para encapsular:

- Controlar el uso.
- Minimizar los efectos del cambio.

### La encapsulación permite el control

Controlar el uso es el primer motivo para encapsular. Cuando conducimos un coche pensamos únicamente en el acto de conducir, no en los aspectos internos del coche. Si retiramos dinero de una cuenta, no nos planteamos cómo estará representada. La encapsulación y los métodos de comportamiento sirven para diseñar objetos de software que funcionan sólo como queremos que lo hagan.

### La encapsulación permite el cambio

La segunda razón para encapsular es una consecuencia de la primera. Si el mecanismo de creación de un objeto es privado, es posible modificarlo de forma que los cambios no afecten directamente a los usuarios del objeto (que sólo pueden acceder a los métodos públicos). Esto puede ser extremadamente útil en la práctica, ya que los nombres de los métodos se suelen fijar mucho antes de su realización.

La capacidad de realizar cambios internos está muy relacionada con la abstracción. Dados dos diseños para una misma clase, por regla general hay que elegir el que tenga menos métodos públicos.

En otras palabras: Si se puede elegir entre hacer un método público o privado, lo mejor es hacerlo privado. Un método privado se puede cambiar libremente y más tarde, tal vez, convertirlo en un método público. Por el contrario, no es posible convertir un método público en privado sin romper el código cliente.

## Datos de objetos

**Objetivo del tema**

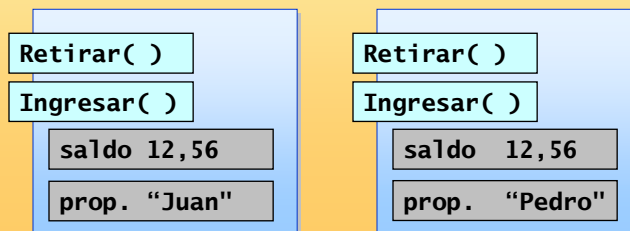
Discutir aspectos relacionados con datos de objetos privados.

**Explicación previa**

¿Es necesario que objetos de la misma clase compartan sus datos? Normalmente no lo hacen.

■ Los datos de objetos describen información para objetos *concretos*

- Por ejemplo, cada cuenta bancaria tiene su propio saldo. Si dos cuentas tienen el mismo saldo, será sólo una coincidencia .



La mayor parte de los datos dentro de un objeto describen información sobre ese objeto concreto. Por ejemplo, cada cuenta bancaria tiene su propio saldo. Por supuesto, es perfectamente posible que haya muchas cuentas bancarias con el mismo saldo, pero eso será sólo una coincidencia.

Un dato de un objeto es privado y sólo los métodos del objeto pueden acceder a él. Esta encapsulación y separación significa que, en la práctica, un objeto es una entidad autónoma.

## Uso de datos estáticos

### Objetivo del tema

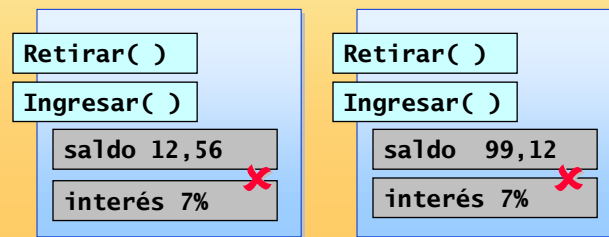
Explicar cómo los objetos pueden compartir datos privados.

### Explicación previa

Ocasionalmente es necesario que objetos de la misma clase compartan datos.

### ■ Los datos estáticos describen información para *todos* los objetos de una clase

- Por ejemplo, supongamos que todas las cuentas comparten el mismo interés. No sería conveniente almacenar el interés en todas las cuentas. ¿Por qué?



Hay veces en que no tiene sentido almacenar información dentro de cada objeto. Si todas las cuentas bancarias comparten siempre el mismo interés, por ejemplo, almacenar el interés dentro de cada objeto cuenta no sería conveniente por los siguientes motivos:

### Recomendación al profesor

El interés se muestra en la transparencia como dato de objeto. La "X" roja indica que no es una buena idea.

El segundo ejercicio de la práctica pide a los estudiantes que creen datos y métodos estáticos.

- Es una forma poco eficaz de resolver el problema tal como está planteado: “Todas las cuentas bancarias comparten el mismo interés.”
- Aumenta innecesariamente el tamaño de cada objeto, utilizando más memoria cuando se ejecuta el programa y más espacio en disco cada vez que se guarda.
- Hace difícil cambiar el interés, ya que habría que modificarlo en todos y cada uno de los objetos cuenta y eso podría hacer imposible acceder a las cuentas durante el cambio.
- Aumenta el tamaño de la clase. El dato privado de interés necesitaría métodos públicos, por lo que la clase cuenta se haría menos compacta y no funcionaría todo lo bien que debiera.

Para resolver este problema no se debe compartir a nivel de objeto información que sea común a varios objetos. En lugar de describir el interés muchas veces a nivel de objeto, es mucho mejor describirlo una sola vez a nivel de clase. El interés definido a nivel de clase se convierte en la práctica en un dato global.

Pero los datos globales, por definición, no se almacenan dentro de una clase, y por lo tanto no pueden ser encapsulados. A causa de esto, muchos lenguajes de programación orientados a objetos (incluyendo C#) no permiten los datos globales. En su lugar permiten describir datos como estáticos.

## Declaración de datos estáticos

Un dato estático (*static*) se define físicamente dentro de una clase (que es una entidad estática de tiempo de compilación) y se beneficia de la encapsulación de ésta, pero lógicamente está asociado a la clase y no a un objeto concreto. En otras palabras, un dato estático se declara dentro de una clase por conveniencia sintáctica y existe aunque el programa nunca cree objetos de esa clase.

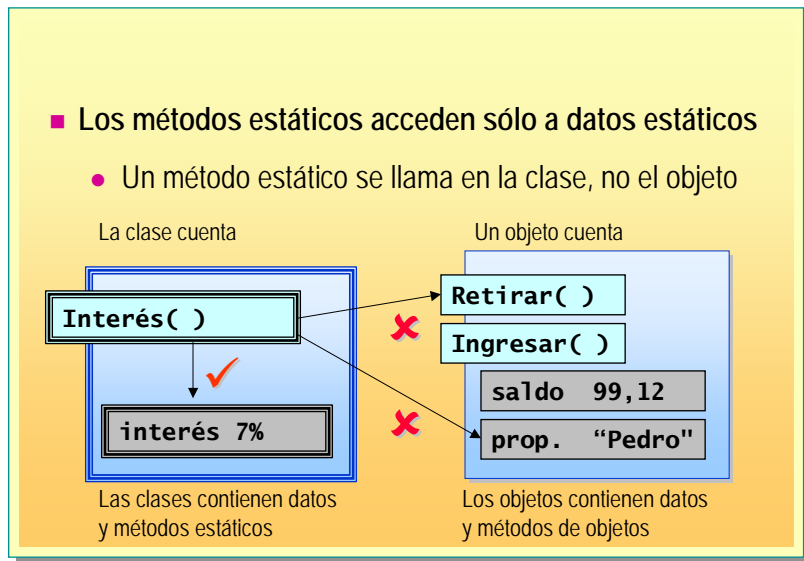
## Uso de métodos estáticos

### Objetivo del tema

Describir los métodos estáticos.

### Explicación previa

Si el interés pertenece a la clase cuenta en lugar de a un objeto cuenta concreto, y si se utiliza encapsulación para ocultar la representación interna del interés, ¿cómo se puede acceder al interés o modificarlo?



Para encapsular datos estáticos se pueden utilizar métodos estáticos. En el ejemplo de la transparencia, el interés pertenece a la clase cuenta y no a un objeto cuenta concreto. Por lo tanto, tiene sentido que a nivel de clase haya métodos que se puedan emplear para acceder al interés o modificarlo.

Es posible declarar métodos como estáticos de la misma forma que se declara un dato estático. Los métodos estáticos existen a nivel de clase. La accesibilidad de métodos y datos estáticos se puede controlar con modificadores como *public* (público) y *private* (privado). Métodos estáticos públicos con datos estáticos privados permiten encapsular datos estáticos del mismo modo que se encapsulan datos de objetos.

Un método estático existe a nivel de clase y las llamadas a ese método se hacen con referencia a la clase, no a un objeto. Esto significa que un método estático no puede utilizar el operador **this**, que implícitamente apunta al objeto que hace una llamada a un método de objeto. En otras palabras, un método estático no puede acceder a datos o métodos que no sean estáticos. Los únicos miembros de una clase a los que puede acceder un método estático son datos estáticos y otros métodos estáticos.

Los métodos estáticos siguen teniendo acceso a todos los miembros privados de una clase y pueden acceder a datos privados no estáticos por medio de una referencia a objeto. El siguiente código muestra un ejemplo:

```
class Reloj
{
    ...
    public static void Reset(Reloj t)
    {
        t.hora = 0;    // Okay
        t.minuto = 0;  // Okay
        hora = 0;      // error al compilar
        minuto = 0;    // error al compilar
    }
    private int hora, minuto;
}
```



## ◆ El lenguaje C# y la orientación a objetos

**Objetivo del tema**

Ofrecer una introducción a los temas tratados en esta sección.

**Explicación previa**

Ahora que hemos visto parte de la teoría que hay detrás de la orientación a objetos, vamos a ver cómo se utiliza en C#.

- “Hola, mundo” de nuevo
- Definición de clases simples
- Instancias de nuevos objetos
- Uso del operador `this`

---

En esta sección volveremos a examinar el programa “Hola, mundo” original y explicaremos su estructura desde una perspectiva orientada a objetos.

Al final de esta lección, usted será capaz de:

- Usar los mecanismos que hacen que un objeto pueda crear otro en C#.
- Definir clases anidadas.

## “Hola, mundo” de nuevo

**Objetivo del tema**

Explicar cómo se invoca **Main** cuando se ejecuta una aplicación.

**Explicación previa**

Un programa C# es una clase que tiene un método estático llamado **Main**. ¿Por qué?

```
using System;

class Hello
{
    public static int Main( )
    {
        Console.WriteLine("Hello, World");
        return 0;
    }
}
```

La transparencia muestra el código para “Hola, mundo”. Podemos hacer y contestar varias preguntas:

- ¿Cómo invoca una clase el runtime?
- ¿Por qué **Main** es estático?

### ¿Cómo invoca una clase el runtime?

Si sólo hay un método **Main**, el compilador lo interpretará automáticamente como el punto de entrada al programa. El siguiente código muestra un ejemplo:

```
// UnaEntrada.cs
class UnaEntrada
{
    static void Main( )
    {
        ...
    }
}
// fin del archivo
```

```
c:\> csc UnaEntrada.cs
```

---

**Aviso** El punto de entrada a un programa C# debe ser **Main** con una “M” mayúscula. La signatura de **Main** también es importante.

---

Si, por el contrario, existen varios métodos llamados **Main**, es necesario que uno de ellos esté designado explícitamente como punto de entrada al programa (y que ese **Main** también sea público explícitamente). El siguiente código muestra un ejemplo:

```
// DosEntradas.cs
using System;
class EntradaUno
{
    public static void Main( )
    {
        Console.WriteLine("EntradaUno.Main( )");
    }
}
class EntradaDos
{
    public static void Main( )
    {
        Console.WriteLine("EntradaDos.Main( )");
    }
}
// Fin del archivo

c:\> csc /main:EntradaUno DosEntradas.cs
c:\> dosentradas.exe
EntradaUno.Main( )
c:\> csc /main:EntradaDos DosEntradas.cs
c:\> dosentradas.exe
EntradaDos.Main( )
c:\>
```

La opción de línea de comandos distingue mayúsculas y minúsculas. Si el nombre de la clase que contiene **Main** es **EntradaUno** (con E y O mayúsculas), el siguiente comando no funcionará:

```
c:\> csc /main:entradauno DosEntradas.cs
```

No se puede crear un programa ejecutable si el proyecto no contiene ningún método **Main**. Sin embargo, es posible crear una biblioteca de vínculos dinámicos (DLL) de la siguiente manera:

```
// SinEntrada.cs
using System;
class SinEntrada
{
    public static void NoMain( )
    {
        Console.WriteLine("SinEntrada.NoMain( )");
    }
}
// Fin del archivo

c:\> csc /target:library SinEntrada.cs
c:\> dir
...
SinEntrada.dll
...
```

## ¿Por qué Main es estático?

El hecho de que **Main** sea estático hace que sea posible invocarlo sin que el runtime tenga que crear una instancia de la clase.

Las llamadas a métodos no estáticos sólo pueden estar en un objeto, como se ve en el siguiente código:

```
class Ejemplo
{
    void NonStatic( ) { ... }
    static void Main( )
    {
        Ejemplo eg = new Example( );
        eg.NonStatic( ); // Compila
        NonStatic( );    // error al compilar
    }
    ...
}
```

Esto significa que si **Main** no es estático, como en el siguiente código, el runtime necesita crear un objeto para hacer una llamada a **Main**.

```
class Ejemplo
{
    void Main( )
    {
        ...
    }
}
```

En otras palabras, en la práctica el runtime tendría que ejecutar el siguiente código:

```
Ejemplo ejecuta = new Example( );
ejecuta.Main( );
```

## Definición de clases simples

### Objetivo del tema

Describir la sintaxis básica para la definición de clases.

### Explicación previa

Ya sabemos qué son las clases. Ahora tenemos que saber cómo se definen.

- Datos y métodos juntos dentro de una clase
- Los métodos son públicos, los datos son privados

```
class BankAccount
{
    public void Withdraw(decimal cantidad)
    { ... }
    public void Deposit(decimal cantidad)
    { ... }
    private decimal balance;
    private string name;
}
```

Métodos públicos  
describen un  
comportamiento  
accesible

Campos privados  
describen un  
estado  
inaccesible

A pesar de ser distintas semánticamente, las clases y estructuras tienen algunos parecidos sintácticos. Para definir una clase en vez de una estructura:

- Se usa la palabra reservada **class** en lugar de **struct**.
- Los datos se declaran dentro de la clase exactamente igual que para una estructura.
- Los métodos se declaran dentro de la clase.
- Se añaden modificadores de acceso a las declaraciones de los datos y métodos. Los dos modificadores de acceso más sencillos son **public** y **private** (los otros tres se discutirán más adelante en este curso).

---

**Nota** El uso de **public** y **private** para forzar la encapsulación es responsabilidad del programador. C# no impide la creación de datos públicos.

---

El significado de *public* es “acceso sin limitaciones”, mientras que *private* quiere decir “acceso limitado al tipo que lo contiene”. El siguiente ejemplo aclara este punto:

```
class CuentaBancaria
{
    public void Ingresar(decimal cantidad)
    {
        saldo += cantidad;
    }
    private decimal saldo;
}
```

En este ejemplo, el método **Ingresar** puede acceder al **saldo** privado porque **Ingresar** es un método de **CuentaBancaria** (el tipo que contiene **saldo**). En otras palabras, **Ingresar** está en la parte de dentro. Desde el exterior nunca es posible acceder a miembros privados. En el siguiente ejemplo no se compilará la expresión `cuentaAtacada.saldo`.

```
class LadronDeBancos
{
    public void RobarDe(CuentaBancaria cuentaAtacada)
    {
        cuentaAtacada.saldo -= 999999M;
    }
}
```

La expresión `cuentaAtacada.saldo` no se compilará porque está dentro del método **RobarDe** de la clase **LadronDeBancos**. Sólo los métodos de la clase **CuentaBancaria** pueden acceder a miembros privados de objetos de **CuentaBancaria**.

Para declarar datos estáticos se sigue el mismo patrón que para métodos estáticos (como **Main**), poniendo la palabra reservada **static** antes de la declaración del campo. El siguiente código muestra un ejemplo:

```
class CuentaBancaria
{
    public void Ingresar(decimal cantidad) { ... }
    public static void Main( ) { ... }
    ...
    private decimal saldo;
    private static decimal interés;
}
```

Un miembro de una clase para el que no especifique ningún modificador de acceso será privado por defecto. En otras palabras, los dos métodos siguientes son idénticos semánticamente:

```
class CuentaBancaria
{
    ...
    decimal saldo;
}

class CuentaBancaria
{
    ...
    private decimal saldo;
}
```

---

**Consejos** Aunque no sea estrictamente necesario, se considera una buena práctica de programación escribir **private** explícitamente.

El orden en que se declaran los miembros de una clase no tiene importancia para el compilador de C#. Sin embargo, se considera una buena práctica de programación declarar los miembros públicos (métodos) antes que los privados (datos). Esto se debe a que el usuario de una clase sólo tiene acceso a los miembros públicos, por lo que declarar estos antes que los privados refleja esa prioridad.

---

## Instancias de nuevos objetos

### Objetivo del tema

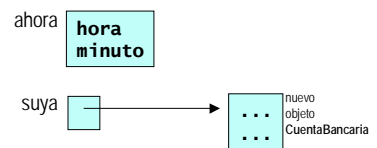
Presentar el operador `new`.

### Explicación previa

Antes de crear un objeto, es preciso crear una instancia de él.

- Al declarar una variable de clase no se crea un objeto
- Para crear un objeto se usa el operador `new`

```
class Program
{
    static void Main( )
    {
        Reloj ahora;
        ahora.hora = 11;
        Cuenta Bancaria suya = new CuentaBancaria( );
        suya.Ingresar(999999M);
    }
}
```



Consideremos los siguientes códigos de ejemplo:

### Recomendación al profesor

El primer ejercicio de la práctica hace hincapié en la diferencia entre estructuras y clases y utiliza el operador `new` para crear objetos.

```
struct Reloj
{
    public int hora, minuto;
}
class Programa
{
    static void Main( )
    {
        Reloj ahora;
        ahora.hora = 11;
        ahora.minuto = 59;
        ...
    }
}
```

Las variables del tipo `struct` son *tipos de valor*. Esto quiere decir que, cuando se declara una variable de estructura (como *ahora* en **Main**), se crea un valor en la pila. En este caso, la estructura *Reloj* contiene dos `ints`, por lo que la declaración de *ahora* crea dos `ints` en la pila, una llamada *ahora.hora* y otra *ahora.minuto*. Estos dos `ints` NO se inicializan a cero por defecto. Por eso no es posible leer el valor de *ahora.hora* ni de *ahora.minuto* hasta que se les asigna un valor. Los valores sólo tienen validez en el bloque en el que se declaran. En este ejemplo, el ámbito de *ahora* es **Main**. Esto significa que cuando el flujo de control abandona **Main** (por un `return` normal o porque se ha lanzado una excepción), *ahora* quedará fuera de ámbito y dejará de existir.



Las clases son totalmente distintas, como muestra el siguiente código:

```
class Reloj // NOTA: Reloj es ahora una clase
{
    public int hora, minuto;
}
class Programa
{
    static void Main( )
    {
        Reloj ahora;
        ahora.hora = 11;
        ahora.minuto = 59;
        ...
    }
}
```

Cuando se declara una variable de clase, no se crea una instancia u objeto de esa clase. En este caso, la declaración de *ahora* no crea un objeto de la clase **Reloj**. Al declarar una variable de clase se crea una referencia que puede apuntar a un objeto de esa clase. Por esta razón las clases se llaman *tipos de referencia*. Esto significa que, si el runtime ejecutara el código anterior, intentaría acceder a los enteros dentro de un objeto *Reloj* que no existe. Afortunadamente, el compilador avisaría de este error con el siguiente mensaje:

error CS0165: Uso de variable local no asignada 'ahora'

Para corregir este error es necesario crear un objeto *Reloj* (usando la palabra reservada **new**) y hacer que la variable de referencia *ahora* apunte al objeto recién creado, como en el siguiente código:

```
class Programa
{
    static void Main( )
    {
        Reloj ahora = new Reloj( );
        ahora.hora = 11;
        ahora.minuto = 59;
        ...
    }
}
```

Recordemos que, cuando se crea un valor de estructura local en la pila, los campos NO se inicializan a cero por defecto. Las clases son diferentes: cuando se crea un objeto como una instancia de una clase, como anteriormente, los campos del objeto se inicializan a cero por defecto. El siguiente código se compilará sin errores:

```
class Programa
{
    static void Main( )
    {
        Reloj ahora = new Reloj( );
        Console.WriteLine(ahora.hora);    // escribe 0
        Console.WriteLine(ahora.minuto);  // escribe 0
        ...
    }
}
```

## Uso de la palabra reservada **this**

**Objetivo del tema**

Describir el operador **this**.

**Explicación previa**

Los métodos dentro de una clase tienen que poder apuntar a la instancia actual de la clase.

- La palabra reservada **this** apunta al objeto usado para la llamada al método
- Es útil en caso de conflicto entre identificadores de distintos ámbitos

```
class CuentaBancaria
{
    ...
    public void PoneNombre(string nombre)
    {
        this.nombre = nombre;
    }
    private string nombre;
}
```

Si esta instrucción fuera `nombre = nombre;` ¿qué ocurriría?

La palabra reservada **this** apunta implícitamente al objeto que está haciendo una llamada a un método de objeto.

En el siguiente código, la instrucción `nombre = nombre` no tendría ningún efecto, ya que el identificador *nombre* en el lado izquierdo de la asignación no se corresponde al campo privado de **CuentaBancaria** llamado *nombre*. Los dos identificadores se refieren al parámetro del método, que también se llama *nombre*.

```
class CuentaBancaria
{
    public void PoneNombre(string nombre)
    {
        nombre = nombre;
    }
    private string nombre;
}
```

---

**Aviso** El compilador de C# *no* emite un aviso para alertar de este error.

---

## Uso de la palabra reservada **this**

Este problema de referencia se puede resolver con la palabra reservada **this**, como se ve en la transparencia. La palabra reservada **this** apunta al objeto actual para el que se hace la llamada al método.

---

**Nota** Los métodos estáticos no pueden usar **this**, puesto que no se les llama empleando un objeto.

---

## Cambio del nombre del parámetro

También es posible resolver el problema de referencia cambiando el nombre del parámetro, como en el siguiente ejemplo:

```
class CuentaBancaria
{
    public void PoneNombre(string nuevoNombre)
    {
        nombre = nuevoNombre;
    }
    private string nombre;
}
```

---

**Consejo** En C# es muy habitual usar **this** para escribir constructores. El siguiente código muestra un ejemplo:

```
struct Reloj
{
    public Reloj(int hora, int minuto)
    {
        this.hora = hora;
        this.minuto = minuto;
    }
    private int hora, minuto;
}
```

---

---

**Consejo** La palabra clave **this** se usa también para encadenar llamadas. En la siguiente clase, los dos métodos devuelven el objeto que ha hecho la llamada:

```
class Libro
{
    public Libro PuneAutor(string autor)
    {
        this.autor = autor;
        return this;
    }
    public Libro PuneTitulo(string titulo)
    {
        this.titulo = titulo;
        return this;
    }
    private string autor, titulo;
}
```

La devolución de **this** permite encadenar llamadas a métodos, como se muestra a continuación:

```
class Uso
{
    static void Cadena(Libro bueno)
    {
        bueno.PuneAutor(" Fowler").PuneTitulo(" Refactoring");
    }
    static void NoCadena(Libro bueno)
    {
        bueno.PuneAutor("Fowler");
        bueno.PuneTitulo (" Refactoring");
    }
}
```

---

**Nota** Un método estático existe a nivel de clase y las llamadas a él se hacen en referencia a la clase y no a un objeto. Esto significa que un método estático no puede usar el operador **this**.

---

## ◆ Definición de sistemas orientados a objetos

**Objetivo del tema**

Ofrecer una introducción a los temas tratados en esta sección.

**Explicación previa**

En esta lección discutiremos los conceptos de herencia y polimorfismo.

- Herencia
- Jerarquías de clases
- Herencia sencilla y múltiple
- Polimorfismo
- Clases base abstractas
- Interfaces

---

En esta lección discutiremos los conceptos de herencia y polimorfismo. En módulos posteriores estudiaremos cómo se utilizan en C# estos conceptos.

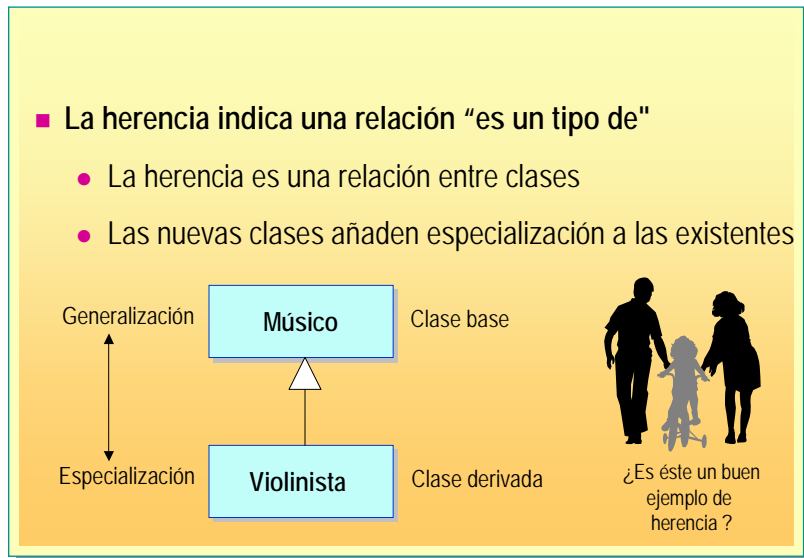
## Herencia

### Objetivo del tema

Explicar el concepto de herencia.

### Explicación previa

La herencia permite definir nuevas clases a partir de otras ya existentes.



### Recomendación al profesor

La transparencia incluye una figura de un hombre, una mujer y una niña pequeña en un triciclo. Utilice esta figura como ejemplo de cómo *no* usar la herencia, tal como se explica en las notas.

La herencia es una relación especificada a nivel de clase para reflejar el hecho de que una clase nueva puede derivar de otra ya existente. En la transparencia, la clase **Violinista** deriva de la clase **Músico**. La clase **Músico** se denomina clase *base* (o, con menos frecuencia, la clase padre o superclase), mientras que la clase **Violinista** recibe el nombre de clase *derivada* (o también la clase hija o subclase). La herencia se indica en notación de lenguaje unificado de modelado (UML). En transparencias posteriores veremos más notación UML.

La herencia es una relación muy poderosa, ya que una clase derivada lo hereda todo de su clase base. Por ejemplo, si la clase base **Músico** contiene un método llamado **AfinaTuInstrumento**, este método se convierte automáticamente en miembro de la clase derivada **Violinista**.

Una clase base puede tener un número arbitrario de clases derivadas. Por ejemplo, de la clase **Músico** podrían derivar otras clases nuevas (como **Flautista** o **Pianista**) que también heredarían automáticamente el método **AfinaTuInstrumento** de la clase base **Músico**.

**Nota** Un cambio en la clase base se convierte automáticamente en un cambio para todas las clases derivadas. Por ejemplo, si se añadiera un campo o tipo **IntrumentoMusical** a la clase base **Músico**, todas las clases derivadas (**Violinista**, **Flautista**, **Pianista**, etc.) adquirirían automáticamente un campo o tipo **IntrumentoMusical**. Un error que aparezca en una clase base se extiende automáticamente a todas las clases derivadas (lo que se conoce como *problema de clase base frágil*).

## La herencia en la programación orientada a objetos

La figura de la transparencia muestra a un hombre, una mujer y una niña pequeña montada en un triciclo. Si el hombre y la mujer son los padres biológicos de la niña, ésta heredará la mitad de sus genes del hombre y la otra mitad de la mujer.

Sin embargo, éste no es un buen ejemplo de herencia de clase. Las clases en este caso son **Hombre** y **Mujer**. Hay dos representantes de la clase **Mujer** (uno de ellas con el atributo **edad** menor que 16) y uno de la clase **Hombre**, pero no hay herencia de clase. La única posibilidad de que en este ejemplo hubiera herencia de clase pasaría por considerar que las clases **Hombre** y **Mujer** comparten una clase base **Persona**.



## Jerarquías de clases

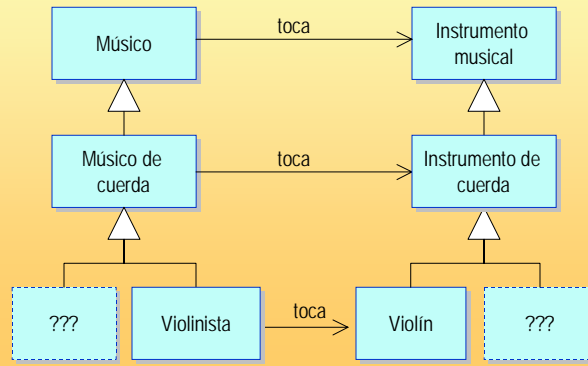
### Objetivo del tema

Explicar las jerarquías de clases.

### Explicación previa

Las clases que derivan de clases base pueden a su vez convertirse en clases base para otras clases.

### ■ Las clases con relaciones de herencia forman jerarquías de clases



Las clases que derivan de clases base pueden a su vez convertirse en clases base para otras clases. En la transparencia, por ejemplo, la clase **MúsicoDeCuerda** deriva de la clase **Músico**, pero a su vez es una clase base para la clase derivada **Violinista**. Un grupo de clases con relaciones de herencia forma una estructura conocida como *jerarquía de clases*. Estas clases representan conceptos más generales (generalización) a medida que ascendemos en la jerarquía, y más especializados (especialización) a medida que nos movemos hacia abajo.

La profundidad de una jerarquía de clases es el número de niveles de herencia que hay en la jerarquía. Las jerarquías de clases más profundas son más difíciles de usar que las jerarquías de clases poco profundas. La mayor parte de las normas de programación recomiendan limitar la profundidad a entre cinco y siete clases.

La transparencia muestra dos jerarquías de clases paralelas, una para músicos y otra para instrumentos musicales. Crear jerarquías de clases no es fácil, ya que las clases se tienen que diseñar como clases base desde el principio. Las jerarquías de herencia son también la característica principal de los marcos de trabajo o frameworks, que son modelos ampliables sobre los que se crean entidades más complejas.

## Herencia sencilla y múltiple

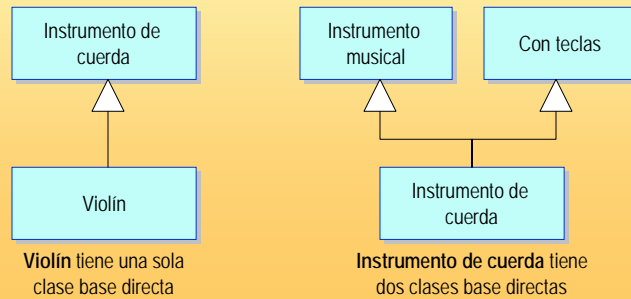
### Objetivo del tema

Comparar la herencia sencilla y la herencia múltiple.

### Explicación previa

En teoría, el comportamiento de una clase puede derivar de más de un antecesor.

- Herencia sencilla: derivadas de una clase base
- Herencia múltiple: derivadas de dos o más clases base



La herencia se denomina sencilla cuando una clase tiene una sola clase base directa. En el ejemplo de la transparencia, la clase **Violín** hereda de una clase, **InstrumentoDeCuerda**, y es un ejemplo de herencia sencilla.

**InstrumentoDeCuerda** deriva de dos clases, pero esto no afecta a la clase **Violín**. La herencia sencilla también puede ser difícil de utilizar. Es bien conocido que la herencia es una de las herramientas más potentes de modelado de software, pero al mismo tiempo una de las peor entendidas y empleadas.

### Recomendación al profesor

Puede utilizar un acordeón como ejemplo de herencia múltiple.

La herencia se denomina múltiple una clase tiene dos o más clases base directas. En el ejemplo de la transparencia, la clase **InstrumentoDeCuerda** deriva directamente de dos clases, **InstrumentoMusical** y **ConTeclas**, y proporciona un ejemplo de herencia múltiple. La herencia múltiple se utiliza mal en un gran porcentaje de casos. C#, como la mayor parte de los lenguajes de programación modernos (excluido C++), limita el uso de la herencia múltiple: está permitida la herencia de un número ilimitado de interfaces, pero sólo se puede heredar de una no interfaz (es decir, una clase abstracto o concreta como mucho). Los términos interfaz, clase abstracta y clase concreta se discutirán más adelante en este módulo.

Todas las formas de herencia, pero especialmente la herencia múltiple, ofrecen muchas vistas del mismo objeto. Por ejemplo, un objeto **Violín** se podría usar a nivel de la clase **Violín**, pero también a nivel de la clase **InstrumentoMusical**.

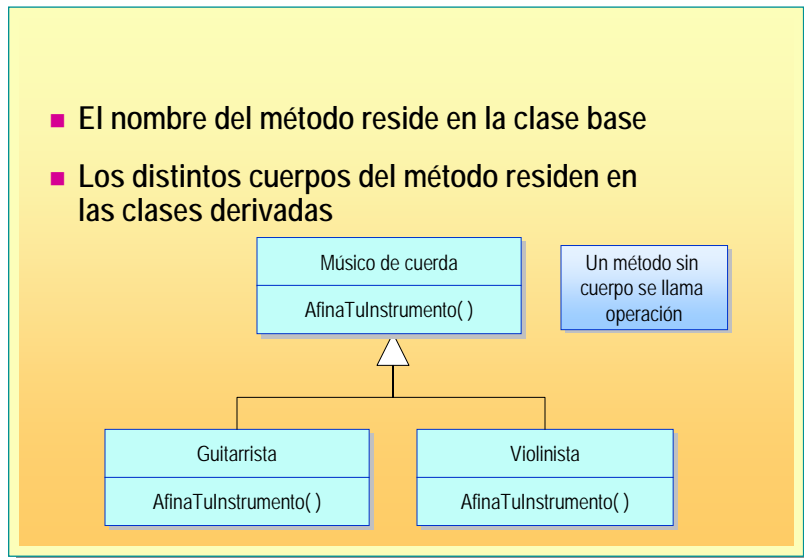
## Polimorfismo

### Objetivo del tema

Explicar el concepto de polimorfismo.

### Explicación previa

A veces puede ser necesario utilizar un método de diferentes maneras en distintas clases hijas.



Polimorfismo significa literalmente *muchas formas*. Es el concepto según el cual un método declarado en una clase base se puede utilizar de muchas formas diferentes en las distintas clases derivadas.

Supongamos que todos los músicos de una orquesta están afinando sus instrumentos antes de un concierto. Sin polimorfismo, el director tiene que ir de músico en músico para ver qué tipo de instrumento toca y darle instrucciones sobre cómo afinarlo. Con polimorfismo, el director sólo tiene que decirle a cada músico “afina tu instrumento”. El instrumento que toque cada músico le es indiferente; sólo necesita saber que todos ellos responderán con el comportamiento solicitado y de una forma adecuada al instrumento que tocan. De esta forma el director no tiene la responsabilidad de saber cómo se afinan todos los distintos tipos de instrumentos, sino que este conocimiento se divide entre los distintos tipos de músicos: un guitarrista sabe cómo afinar una guitarra, un violinista sabe cómo afinar un violín, etc. De hecho, el director no sabe cómo afinar *ninguno* de los instrumentos. Esta asignación descentralizada de responsabilidades significa también que se pueden añadir a la jerarquía nuevas clases derivadas (como **Tambor**) sin que necesariamente se modifiquen el resto de las clases (como el director).

Sólo hay un problema. ¿Cómo es el cuerpo del método al nivel de la clase base? Si no se sabe qué tipo de instrumento es el que toca un músico, es imposible saber cómo afinar ese instrumento. Para hacer frente a este problema, en la clase base sólo se puede declarar el nombre del método, pero no el cuerpo. Un nombre de método que no tenga cuerpo se llama *operación*. Una de las formas de denotar una operación en UML es poniéndola en cursiva, como se muestra en la transparencia.

## Clases base abstractas

### Objetivo del tema

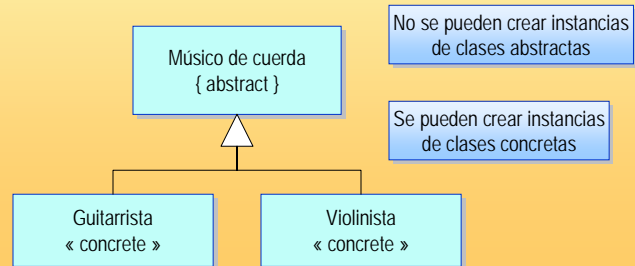
Presentar el concepto de clase abstracta.

### Explicación previa

¿Tendría sentido crear un objeto **Músico de cuerda** en tiempo de ejecución?

### ■ Algunas clases existen sólo para ser clases base

- No tiene sentido crear instancias de estas clases
- Estas clases son *abstractas*



### Recomendación al profesor

El ejemplo de la transparencia es el mismo que el empleado en la transparencia anterior y el mismo que se usará en la siguiente.

En una jerarquía de clases típica, la operación (el nombre de un método) se declara en la clase base y el cuerpo del método se escribe de diferentes maneras en las distintas clases derivadas. La clase base existe únicamente para introducir el nombre del método en la jerarquía. En particular, no es necesario que la operación de la clase base tenga cuerpo. Por ello es fundamental que la clase base no se utilice como una clase normal y, lo que es más importante, que no sea posible crear instancias de la clase base, ya que de lo contrario se correría el riesgo de hacer una llamada a una operación sin cuerpo. Se necesita un mecanismo que haga imposible la creación de instancias de estas clases base: marcar la clase base como abstracta.

En un diseño UML se puede marcar una clase como abstracta escribiendo su nombre en cursiva o poniendo la palabra *abstract* entre llaves ({ y }). Por otra parte, es posible poner la palabra *concrete* o *class* entre dobles paréntesis angulares (<< y >>) para denotar en UML una clase que no es abstracta y que se puede usar para crear instancias. La transparencia ilustra esta notación. Todos los lenguajes de programación orientados a objetos tienen estructuras gramaticales para clases abstractas (incluso C++ puede usar constructores protegidos).

A veces la creación de una clase base abstracta es más retrospectiva y se combinan en una nueva clase base características comunes duplicadas en las clases derivadas. Sin embargo, también en este caso es necesario marcar la clase como abstracta, ya que su propósito no es la creación de instancias, sino servir únicamente de clase base.

## Interfaces

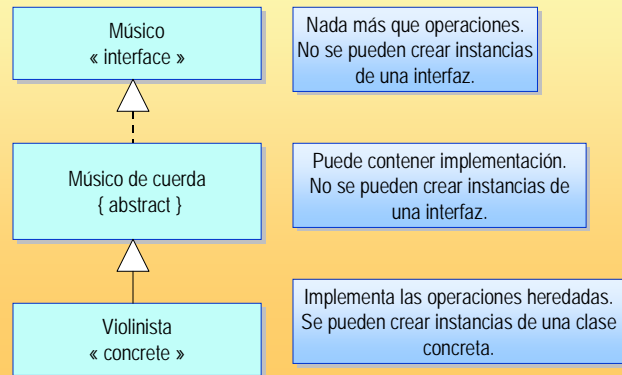
### Objetivo del tema

Describir el propósito de las interfaces.

### Explicación previa

¿Cómo es posible garantizar que todos los músicos (sea cual sea su tipo) derivados de **Músico de cuerda** saben realmente cómo afinar o tocar sus instrumentos?

### ■ Las interfaces contienen sólo operaciones, no implementación



### Recomendación al profesor

Hay dos puntos que vale la pena discutir en la pizarra. El primero es la importancia de que haya una interfaz por encima de una clase abstracta. El segundo se refiere al hecho de que en realidad hay dos especies diferentes de clases abstractas: La primera, que es pública, participa en la jerarquía y tiene una interfaz por encima, mientras que la segunda, que es privada, no tiene participación lógica en la jerarquía y normalmente se utiliza sólo para modificar el código y evitar repeticiones.

Las clases abstractas y las interfaces se parecen en que ninguna de ellas se puede usar para crear instancias de objetos. Sin embargo, se diferencian en que una clase abstracta puede contener métodos con cuerpo mientras que una interfaz no contiene ninguno, sino sólo operaciones (nombres de métodos). Podríamos decir que una interfaz es todavía más abstracta que una clase abstracta.

En UML se puede definir una interfaz poniendo la palabra *interface* entre dobles paréntesis angulares (<< y >>). Todos los lenguajes de programación orientados a objetos tienen estructuras gramaticales para interfaces.

Las interfaces son importantes en los programas orientados a objetos y UML tiene una notación y una terminología específicas para ellas. Se dice que se *implementa* una interfaz cuando se hace una derivación desde ella; en UML, esto se indica con una línea discontinua llamada *realización*. Se dice que se *extiende* una clase cuando se hace una derivación desde una no interfaz (una clase abstracta o una clase concreta); en UML, esto se indica con una línea continua llamada *generalización/especialización*.

Las interfaces deben estar en lo más alto de una jerarquía de clases. La idea es sencilla: Si se puede programar para una interfaz (es decir, si sólo se usan aquellas características de un objeto que están declaradas en su interfaz), el programa pierde totalmente la dependencia de ese objeto específico y su clase concreta. En otras palabras, cuando se programa para una interfaz se pueden usar indistintamente muchos objetos diferentes de muchas clases distintas. En la programación orientada a objetos, esta capacidad de hacer cambios sin que causen efectos secundarios está en el origen de la máxima "Programa para una interfaz y no para un código".