Herencia en C#

Contenido

Notas para el instructor

Este módulo proporciona a los estudiantes información detallada sobre la herencia de clases en C# y explica cómo derivar nuevas clases a partir de otras ya existentes. También discute el uso de los tipos de métodos **virtual**, **override** y **new**. Explica brevemente las clases selladas y describe conceptos de interfaces. Los estudiantes aprenderán a declarar una interfaz y a implementar los dos tipos de métodos de interfaz. Finalmente, el método discute las clases abstractas y explica cómo implementar métodos y clases abstractas en una jerarquía de clases.

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

- Derivar una clase nueva a partir de una clase base y hacer llamadas a miembros y constructores de la clase base desde la clase derivada.
- Declarar métodos como **virtual**es y sustituirlos (**override**) u ocultarlos, según las necesidades.
- Sellar una clase para que de ella no pueda derivar ninguna otra.
- Implementar interfaces de forma implícita y explícita.
- Describir el uso de clases abstractas y su implementación de interfaces.

Descripción general

Objetivo del tema
Ofrecer una introducción a los contenidos y objetivos del módulo.

Explicación previa
En este módulo estudiaremos la herencia de clases en C#.

- Derivación de clases
- Implementación de métodos
- Uso de clases selladas
- Uso de interfaces
- Uso de clases abstractas

En un sistema orientado a objetos, la herencia es la capacidad de un objeto de heredar datos y funcionalidad de su objeto padre. De esta forma, el objeto padre puede ser sustituido por un objeto hijo. La herencia también permite crear nuevas clases a partir de otras ya existentes, en lugar de crearlas partiendo de cero, y añadir luego el código que sea necesario para la nueva clase. La clase padre en la que está basada la nueva clase se llama *clase base*, mientras que la clase hija se conoce como *clase derivada*.

Cuando se crea una clase derivada hay que tener en cuenta que puede sustituir al tipo de clase base. Esto significa que la herencia no sólo es un mecanismo para la reutilización de código, sino también un mecanismo de clasificación de tipos. Este último aspecto es más importante que el primero.

En este módulo aprenderemos a derivar una clase de una clase base. También veremos cómo implementar métodos en una clase derivada, definiéndolos como métodos virtuales en la clase base y sustituyéndolos u ocultándolos, según el caso, en la clase derivada. Aprenderemos a sellar una clase para que no pueda derivar de ella ninguna otra. Finalmente, estudiaremos cómo implementar interfaces y clases abstractas, que definen las condiciones de un contrato al que están sujetas las clases derivadas.

Al final de este módulo, usted será capaz de:

- Derivar una clase nueva a partir de una clase base y hacer llamadas a miembros y constructores de la clase base desde la clase derivada.
- Declarar métodos como **virtual**es y sustituirlos (**override**) u ocultarlos, según las necesidades.
- Sellar una clase para que de ella no pueda derivar ninguna otra.
- Implementar interfaces de forma implícita y explícita.

- Describir el uso de clases abstractas y su implementación de interfaces.

◆ Derivación de clases

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En esta sección discutiremos cómo derivar una clase de una clase base.

- Extensión de clases base
- Acceso a miembros de la clase base
- Llamadas a constructores de la clase base

Sólo es posible derivar una clase a partir de una clase base si ésta ha sido diseñada para permitir la herencia. Esto se debe a que los objetos deben tener la estructura adecuada, ya que de lo contrario la herencia no resultará eficaz. Este hecho tiene que quedar claro en una clase base que esté diseñada para herencia. Si se deriva una nueva clase de una clase base que no está bien diseñada, cualquier cambio futuro en la clase base podría hacer que la clase derivada fuese inutilizable.

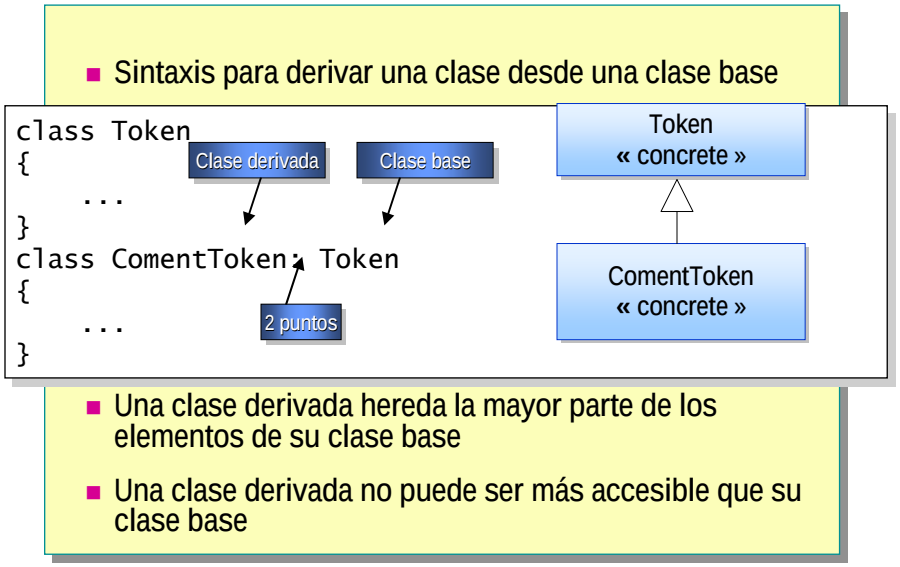
Al final de esta lección, usted será capaz de:

- Derivar una clase nueva a partir de una clase base.
- Acceder a los miembros y constructores de la clase base desde la clase derivada.

Extensión de clases base

Objetivo del tema
Explicar el procedimiento para extender una clase base.

Explicación previa
Es fácil entender la sintaxis de C# para derivar una clase de otra.



La derivación de una clase desde una clase base se conoce también como *extensión* de la clase base. Una clase C# se puede extender como máximo una clase.

Sintaxis para la derivación de una clase

Para indicar que una clase deriva de otra se emplea la siguiente sintaxis:

```
class Derived: Base
{
    ...
}
```

Los elementos de esta sintaxis se indican en la transparencia. Cuando se declara una clase derivada, la clase base se especifica después de dos puntos. Los blancos antes y después de los dos puntos no tienen importancia. El estilo recomendado para esta sintaxis es no poner espacios antes de los dos puntos y poner un espacio después de ellos.

Herencia de la clase derivada

Una clase derivada hereda todo de su clase base, salvo los constructores y destructores. Los miembros públicos de la clase base se convierten implícitamente en miembros públicos de la clase derivada. Sólo los miembros de la clase base tienen acceso a los miembros privados de esta clase, aunque la clase derivada también los hereda.

Accesibilidad de una clase derivada

Una clase derivada no puede ser más accesible que su clase base. Por ejemplo, no es posible derivar una clase pública de una clase privada, como se ve en el siguiente código:

```
class Example
{
    private class NestedBase { }
    public class NestedDerived: NestedBase { } // Error
}
```

La sintaxis de C# para derivar una clase de otra también está permitida en C++, donde indica implícitamente una relación de herencia privada a entre la clase base y la derivada. C# no incluye herencia privada; toda la herencia es pública.

Acceso a miembros de la clase base

Objetivo del tema
Explicar la herencia con protección.

Explicación previa
Al igual que otros lenguajes de programación orientados a objetos, C# tiene modificador de acceso **protected** (protegido), además de **public** y **private**.

```
class Token
{   ...
    protected string name;
}
class ComentToken: Token
{   ...
    public string Name( )
    {
        return name;
    }
}
```

```
class Outside
{
    void Fails(Token t)
    {
       ...
      t.name
       ...
  }
}
```

- Los miembros heredados con protección están implícitamente protegidos en la clase derivada
- Los miembros de una clase derivada sólo pueden acceder a sus miembros heredados con protección
- En una struct no se usa el modificador de acceso protected

El significado del modificador de acceso **protected** depende de la relación entre la clase que tiene el modificador y la clase que intenta acceder a los miembros que usan el modificador.

Los miembros de una clase derivada pueden acceder a todos los miembros protegidos de su clase base. Para una clase derivada, la palabra reservada **protected** es equivalente a la palabra **public**. En el fragmento de código de la transparencia, el método **Name** de **ComentToken** puede acceder a la cadena *nombre*, que está protegida dentro de **Token**, porque **Token** es la clase base de **ComentToken**.

Entre dos clases que no tengan una relación base-derivada, por el contrario, los miembros protegidos de una clase se comportan como miembros privados para la otra clase. En el otro fragmento de código de la transparencia, el método **Fails** de **Outside** no puede acceder a la cadena *nombre*, que está protegida dentro de **Token**, porque **Token** no está especificada como clase base de **Outside**.

Miembros heredados con protección

Cuando una clase derivada hereda un miembro con protección, ese miembro también es implícitamente un miembro protegido de la clase derivada. Esto significa que todas clases que deriven directa o indirectamente de la clase base pueden acceder a los miembros protegidos, como se muestra en el siguiente ejemplo:

```
class Base
{
    protected string name;
}

class Derived: Base
{
}

class FurtherDerived: Derived
{
    void Compiles( )
    {
        Console.WriteLine(name); // Okay
    }
}
```

Miembros protegidos y métodos

Los métodos de una clase derivada sólo tienen acceso a sus propios miembros heredados con protección. No pueden acceder a los miembros protegidos de la clase base a través de referencias a ésta. Por ejemplo, el siguiente código generará un error:

```
class ComentToken: Token
{
    void Fails (Token t)
    {
        Console.WriteLine(t.name); // Error al compilar
    }
}
```

Consejo Muchas guías de programación recomiendan mantener todos los datos privados y usar acceso protegido sólo para métodos.

Miembros protegidos y structs

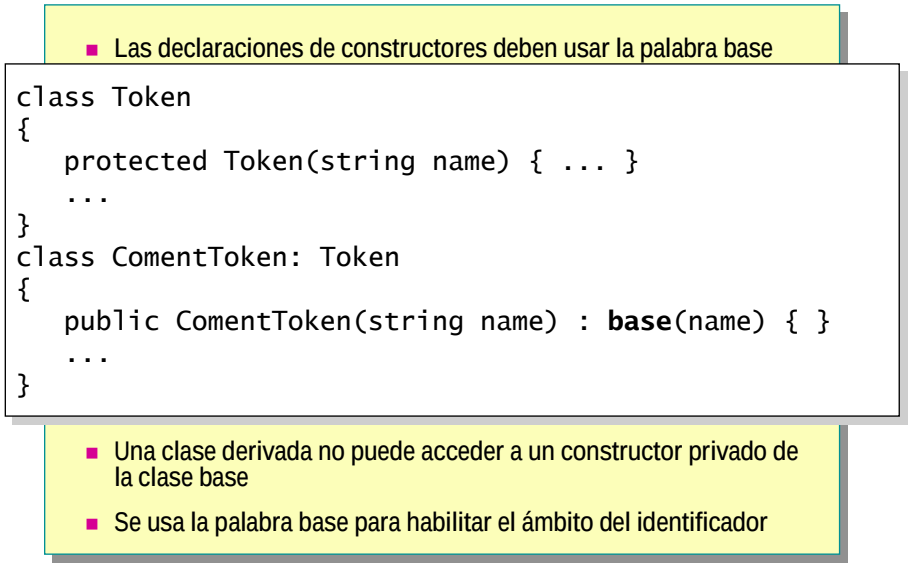
Una **struct** no permite la herencia. Como consecuencia no se puede derivar de una **struct** y, por lo tanto, no es posible usar el modificador de acceso **protected** en una **struct**. Por ejemplo, el siguiente código generará un error:

```
struct Base
{
    protected string name; // Error al compilar
}
```

Llamadas a constructores de la clase base

Objetivo del tema
Explicar cómo hacer llamadas a los constructores de la clase base.

Explicación previa
C# tiene una palabra reservada para llamar a un constructor de la clase base.



Para hacer una llamada a un constructor de la clase base desde un constructor de la clase derivada se usa la palabra reservada **base**, que tiene la siguiente sintaxis:

```
C(...): base( ) {...}
```

El conjunto formado por los dos puntos y la llamada al constructor de la clase base recibe el nombre de *inicializador del constructor*.

Declaraciones de constructores

Si la clase derivada no hace una llamada explícita a un constructor de la clase base, el compilador de C# usará implícitamente un inicializador de constructor de la forma `:base( )`. Esto implica que una declaración de constructor de la forma

```
C(...) {...}
```

es equivalente a

```
C(...): base( ) {...}
```

Este comportamiento implícito es válido en muchos casos porque:

- Una clase sin clases base explícitas extiende implícitamente la clase **System.Object**, que contiene un constructor público sin parámetros.
- Si una clase no contiene ningún constructor, el compilador utilizará inmediatamente un constructor público sin parámetros llamado constructor por defecto.

El compilador no creará un constructor por defecto si una clase tiene su propio constructor explícito. No obstante, el compilador generará un mensaje de error si el constructor indicado no coincide con ningún constructor de la clase base, como se ve en el siguiente código:

```
class Token
{
    protected Token(string name) { ... }
}

class ComentToken: Token
{
    public ComentToken(string name) { ... } // Error aquí
}
```

El error se debe a que el constructor **ComentToken** contiene de forma implícita un inicializador de constructor `:base( )`, pero la clase base **Token** no contiene ningún constructor sin parámetros. El código mostrado en la transparencia permite corregir este error.

Reglas de acceso a constructores

Las reglas de acceso de un constructor derivado a un constructor de la clase base son exactamente las mismas que para los métodos normales. Por ejemplo, si el constructor de la clase base es privado, la clase derivada no tendrá acceso a él:

```
class NoDerivable
{
    private NoDerivable( ) { ... }
}

class Imposible: NoDerivable
{
    public Impossible( ) { ... } // Error al compilar
}
```

En este caso, es imposible que una clase derivada pueda hacer una llamada al constructor de la clase base.

Ámbito de un identificador

También es posible utilizar la palabra reservada **base** para habilitar el ámbito de un identificador. Esto puede resultar útil, puesto que una clase derivada puede declarar miembros con los mismos nombres que miembros de la clase base. El siguiente código muestra un ejemplo:

```
class Token
{
    protected string name;
}
class ComentToken: Token
{
    public void Method(string name)
    {
        base. name = name;
    }
}
```

Nota Al contrario de lo que ocurre en C++, no se utiliza el nombre de la clase base (como **Token** en el ejemplo de la transparencia). La palabra reservada **base** hace referencia claramente a la clase base porque, en C#, una clase puede extender como máximo una clase.

◆ Implementación de métodos

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En esta lección veremos cómo implementar métodos en clases derivadas.

- Definición de métodos virtuales
- Uso de métodos virtuales
- Sustitución de métodos (override)
- Uso de métodos override
- Uso de new para ocultar métodos
- Uso de la palabra reservada new

Los métodos de una clase base se pueden redefinir en una clase derivada si han sido diseñados para permitir la sustitución (override).

Al final de esta lección, usted será capaz de:

- Usar el tipo de método **virtual**.
- Usar el tipo de método **override**.
- Usar el tipo de método **hide**.

Definición de métodos virtuales

Objetivo del tema
Explicar cómo implementar métodos virtuales.

Explicación previa
Los métodos virtuales se pueden emplear para hacer que las clases sean polimórficas.

- Sintaxis: Se declara como virtual

```
class Token
{
    ...
    public int LineNumber( )
    { ...
    }
    public virtual string Name( )
    { ...
    }
}
```

- Los métodos virtuales son polimórficos

Un método virtual especifica *una* implementación de un método que puede ser sustituida por polimorfismo en una clase derivada. Del mismo modo, un método no virtual especifica la *única* implementación de un método. No es posible sustituir por polimorfismo un método no virtual en una clase derivada.

Nota En C#, el hecho de que una clase contenga o no un método virtual es una buena indicación de si el autor la ha diseñado para que sea utilizada como clase base.

Sintaxis de la palabra reservada

Para declarar un método virtual se usa la palabra clave **virtual**, cuya sintaxis se muestra en la transparencia.

Recomendación al profesor
El ejercicio de la práctica combinará el contenido de los tres temas sobre las palabras reservadas **virtual**, **override** y **new**.

Un método virtual debe contener un cuerpo cuando se declara. De lo contrario, el compilador generará un error:

```
class Token
{
    public virtual string Name( ); // Error al compilar
}
```

Uso de métodos virtuales

Objetivo del tema
Describir las restricciones de los métodos virtuales.

Explicación previa
Vamos a estudiar con más detalle los métodos virtuales.

- Para usar métodos virtuales:
  - No se puede declarar métodos virtuales como estáticos
  - No se puede declarar métodos virtuales como privados

Para usar eficazmente los métodos virtuales hay que tener en cuenta lo siguiente:

- No se puede declarar métodos virtuales como estáticos.

  Los métodos virtuales no pueden ser estáticos porque en ese caso serían métodos de clase y el polimorfismo se aplica a objetos, no a clases.

- No se puede declarar métodos virtuales como privados.

  Los métodos virtuales no pueden ser privados porque en ese caso no podrían ser sustituidos por polimorfismo en una clase derivada. A continuación se muestra un ejemplo:

```
class Token
{
    private virtual string Name( ) { ... }
    // Error al compilar
}
```

Sustitución de métodos (override)

Objetivo del tema
Explicar cómo sustituir métodos.

Explicación previa
Se pueden sustituir los métodos que estén declarados como virtuales en la clase base.

- Sintaxis: Se usa la palabra reservada override

```
class Token
{   ...
    public virtual string Name( ) { ... }
}
class ComentToken: Token
{   ...
    public override string Name( ) { ... }
}
```

Un método override especifica *otra* implementación de un método virtual. Los métodos virtuales definidos en una clase base pueden ser sustituidos por polimorfismo en una clase derivada.

Sintaxis de la palabra reservada

Para declarar un método override se usa la palabra clave **override**, como se ve en el siguiente código:

```
class Token
{   ...
    public virtual string Name( ) { ... }
}
class ComentToken: Token
{   ...
    public override string Name( ) { ... }
}
```

Como ocurre con los métodos virtuales, un método override debe contener un cuerpo cuando se declara, ya que de lo contrario el compilador genera un error:

```
class Token
{
    public virtual string Name( ) { ... }
}
class ComentToken: Token
{
    public override string Nombre( ); // Error al compilar
}
```

Uso de métodos override

Objetivo del tema
Describir las restricciones de los métodos override.

Explicación previa
El uso de los métodos override si rige por ciertas reglas.

- Sólo se sustituyen métodos virtuales heredados idénticos

```
class Token
{   ...
    public int LineNumber( )  { ... }
    public virtual string Name( ) { ... }
}
class ComentToken: Token
{   ...
    public override int LineNumber( ) { ... }
    public override string Name( ) { ... }
}
```

- Un método override debe coincidir con su método virtual asociado
- Se puede sustituir un método override
- No se puede declarar explícitamente un override como virtual
- No se puede declarar un método override como static o private

Para usar eficazmente los métodos override hay que tener en cuenta las siguientes restricciones importantes:

- Sólo se pueden sustituir métodos virtuales heredados idénticos.
- Un método override debe coincidir con su método virtual asociado.
- Se puede sustituir un método override.
- No se puede declarar explícitamente un método override como virtual.
- No se puede declarar un método override como static o private.

A continuación se describe con más detalle cada una de estas restricciones.

Sólo se pueden sustituir métodos virtuales heredados idénticos

Un método override se puede emplear para sustituir únicamente un método virtual heredado idéntico. En el código de la transparencia, el método **LineNumber** en la clase derivada **ComentToken** causa un error de compilación porque el método heredado **Token.LineNumber** no está marcado como virtual.

Recomendación al profesor
Las palabras aquí empleadas han sido elegidas cuidadosamente: Un método override debe sustituir a un método virtual heredado idéntico.

Un método override debe coincidir con su método virtual asociado

Una declaración de override debe ser en todo punto idéntica al método virtual al que sustituye. Deben tener el mismo nivel de acceso, el mismo tipo de retorno, el mismo nombre y los mismos parámetros.

Por ejemplo, en el siguiente ejemplo la sustitución falla porque los niveles de acceso son diferentes (protected frente a public), los tipos de retorno cambian (**string** frente a **void**) y los parámetros son distintos (**none** frente a **int**):

```
class Token
{
    protected virtual string Name( ) { ... }
}
class ComentToken: Token
{
    public override void Name(int i) { ... } // Errores
}
```

Se puede sustituir un método override

Un método override es virtual de forma implícita y por tanto se puede sustituir, como se ve en el siguiente ejemplo:

```
class Token
{
    public virtual string Name( ) { ... }
}
class ComentToken: Token
{
    public override string Name( ) { ... }
}
class OneLineCommentToken: ComentToken
{
    public override string Name( ) { ... }   // Okay
}
```

No se puede declarar explícitamente un método override como virtual

Un método override es virtual de forma implícita, pero no puede estar declarado explícitamente como virtual, como se ve en el siguiente ejemplo:

```
class Token
{
    public virtual string Name( ) { ... }
}
class ComentToken: Token
{
    public virtual override string Name( ) { ... } // Error
}
```

No se puede declarar un método override como static o private

Un método override nunca puede estar declarado como static porque en ese caso sería un método de clase y el polimorfismo se aplica a objetos, no a clases.

Del mismo modo, un método override nunca puede ser private, ya que debe sustituir a un método virtual y éste no puede ser privado.

Uso de new para ocultar métodos

Objetivo del tema
Explicar cómo ocultar métodos heredados.

Explicación previa
Si se declara un método que tiene la misma signatura que un método de una clase base, es posible *ocultar* el método base sin utilizar override.

- Sintaxis: Para ocultar un método se usa la palabra reservada new

```
class Token
{   ...
    public int LineNumber( )  { ... }
}
class ComentToken: Token
{   ...
    new public int LineNumber( )  { ... }

}
```

Es posible ocultar un método heredado idéntico introduciendo un nuevo método en la jerarquía de clases. De esta forma, el método original heredado por la clase derivada desde la clase base es sustituido por un método totalmente distinto.

Sintaxis de la palabra reservada

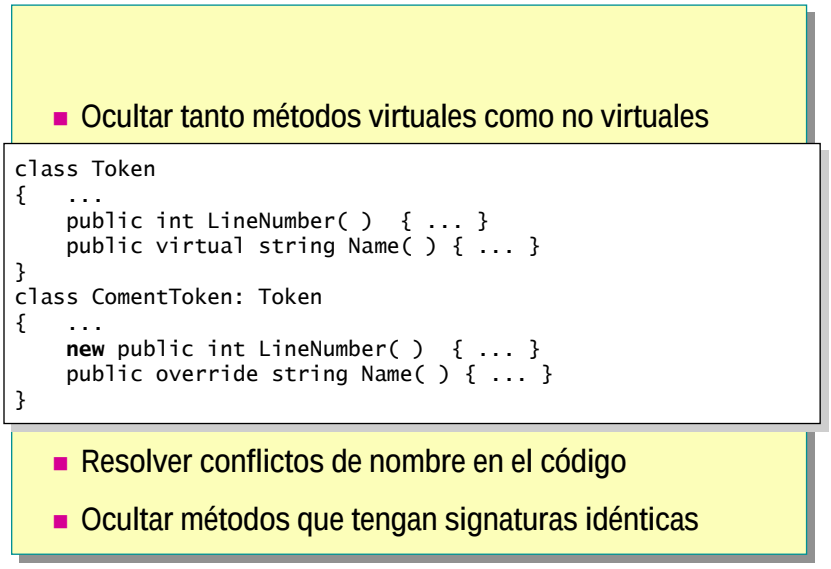
La palabra reservada **new** se emplea para ocultar un método y tiene la siguiente sintaxis:

```
class Token
{   ...
    public int LineNumber( )  { ... }

}
class ComentToken: Token
{   ...
    new public int LineNumber( ) { ... }

}
```

Uso de la palabra reservada new

Objetivo del tema
Explicar cómo usar la palabra reservada **new** para ocultar métodos de forma eficaz.

Explicación previa
Para hacer un buen uso de la palabra reservada **new** hay que tener en cuenta sus características y las restricciones que impone.



El uso de la palabra reservada **new** permite:

- Ocultar tanto métodos virtuales como no virtuales.
- Resolver conflictos de nombre en el código.
- Ocultar métodos que tengan signaturas idénticas.

A continuación se describe con más detalle cada una de estas tareas.

Ocultar tanto métodos virtuales como no virtuales

Emplear la palabra reservada **new** para ocultar un método tiene implicaciones si se hace uso del polimorfismo. En el código de la transparencia, por ejemplo, **ComentToken**.**NumeroLinea** es un método **new** que no tiene ninguna relación con el método **Token**. **LineNumber**. Aunque **Token**. **LineNumber** fuera un método virtual, **ComentToken**. **LineNumber** seguiría siendo un método **new** sin ninguna relación.

En este ejemplo, **ComentToken**. **LineNumber** no es virtual. Esto significa que otra clase derivada no puede sustituir **ComentToken**. **LineNumber**. Sin embargo, el método **new ComentToken**. **LineNumber** se podría declarar como virtual, en cuyo caso otras clases derivadas podrían sustituirlo, como se ve:

Recomendación al profesor
En gran parte, el punto más importante de este tema es el consejo, que hace hincapié en el significado ortogonal del new.

Este tema incluye muchos otros detalles, aunque no es necesario discutir todos ellos en la clase.

```
class ComentToken: Token
{
    ...
    new public virtual int LineNumber ( ) { ... }
}
class OneLineCommentToken: ComentToken
{
    public override int LineNumber ( ) { ... }
}
```

Consejo El estilo recomendado para métodos virtuales con new es

```
new public virtual int LineNumber ( ) { ... }
```

mejor que

```
public new virtual int LineNumber ( ) { ... }
```

Resolver conflictos de nombre en el código

Los conflictos de nombre suelen generar avisos durante la compilación. Consideremos por ejemplo el siguiente código:

```
class Token
{
    public virtual int LineNumber ( ) { ... }
}
class ComentToken: Token
{
    public int LineNumber( ) { ... }
}
```

Al compilar, se recibe un aviso que indica que **ComentToken**. **LineNumber** oculta **Token**. **LineNumber**. Este aviso revela el conflicto de nombres. Se puede elegir entre tres opciones:

1. Añadir un calificador **override** a **ComentToken**. **LineNumber**.
2. Añadir un calificador **new** a **ComentToken**. **LineNumber.** En este caso, el método continúa ocultando el método idéntico en la clase base, pero el **new** explícito comunica al compilador y al personal de mantenimiento del código que el conflicto de nombres es intencionado.
3. Cambiar el nombre del método.

Ocultar métodos que tengan signaturas idénticas

El modificador **new** sólo es necesario si un método de una clase derivada oculta un método visible de la clase base que tiene una signatura idéntica. En el siguiente ejemplo, el compilador avisa de que **new** no es necesario porque los métodos reciben parámetros diferentes y por tanto sus signaturas no son idénticas:

```
class Token
{
    public int LineNumber(short s) { ... }
}
class ComentToken: Token
{
    new public int LineNumber(int i) { ... } // Warning
}
```

Del mismo modo, si dos métodos tienen signaturas idénticas el compilador avisará sobre la posibilidad de usar **new** porque el método de la clase base está oculto. En el siguiente ejemplo, los dos métodos tienen signaturas idénticas porque los tipos de retorno no forman parte de la signatura de un método:

```
class Token
{
    public virtual int LineNumber( ) { ... }
}
class ComentToken: Token
{
    public void LineNumber( ) { ... } // Warning
}
```

Nota La palabra reservada **new** se puede emplear también para ocultar campos y clases anidadas.

◆ Uso de clases selladas

Objetivo del tema
Explicar cómo prevenir la herencia accidental.

Explicación previa
No siempre es conveniente permitir que una clase sea heredada por otra.

- Ninguna clase puede derivar de una clase sellada
- Las clases selladas sirven para optimizar operaciones en tiempo de ejecución
- Muchas clases de .NET Framework son selladas: String, StringBuilder, etc.
- Sintaxis: Se usa la palabra reservada sealed

```
namespace System
{
    public sealed class String
    {
        ...
    }
}
namespace Mine
{
    class FancyString: String { ... }
}
```

No es fácil crear una jerarquía de herencia flexible. La mayor parte de las clases son autónomas y no están diseñadas para que otras clases deriven de ellas. En términos de sintaxis, sin embargo, el procedimiento para la derivación de una clase es muy sencillo y se puede escribir en muy poco tiempo. Esto hace que los programadores puedan caer a veces en la tentación de derivar desde una clase que no está pensada para funcionar como clase base.

Para prevenir este problema, y para que el programador pueda comunicar mejor sus intenciones al compilador y a otros programadores, C# permite declarar una clase como *sealed* (sellada). La derivación de una clase sellada no está permitida.

Sintaxis de la palabra reservada

Para sellar una clase se utiliza la palabra reservada **sealed**, cuya sintaxis es la siguiente:

```
namespace System
{
    public sealed class String
    {
        ...
    }
}
```

Microsoft® .NET Framework contiene muchos ejemplos de clases selladas. La transparencia muestra la clase **System.String**, cuyo alias es la palabra reservada **string**. Esta clase está sellada y, por tanto, ninguna otra clase puede derivar de ella.

Optimización de operaciones en tiempo de ejecución

El modificador **sealed** permite hacer ciertas optimizaciones en tiempo de ejecución. En particular, el hecho de que una clase sellada nunca tenga clases derivadas hace posible transformar llamadas a miembros virtuales de clases selladas en llamadas a miembros no virtuales.

Uso de interfaces

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En esta lección aprenderemos el procedimiento para definir y usar interfaces en C#.

- Declaración de interfaces
- Implementación de varias interfaces
- Implementación de métodos de interfaz

Una interfaz especifica un contrato sintáctico y semántico al que están sujetas todas las clases derivadas. Más concretamente, en ese contrato la interfaz describe el *qué*, mientras que las clases que implementan la interfaz describen el *cómo*.

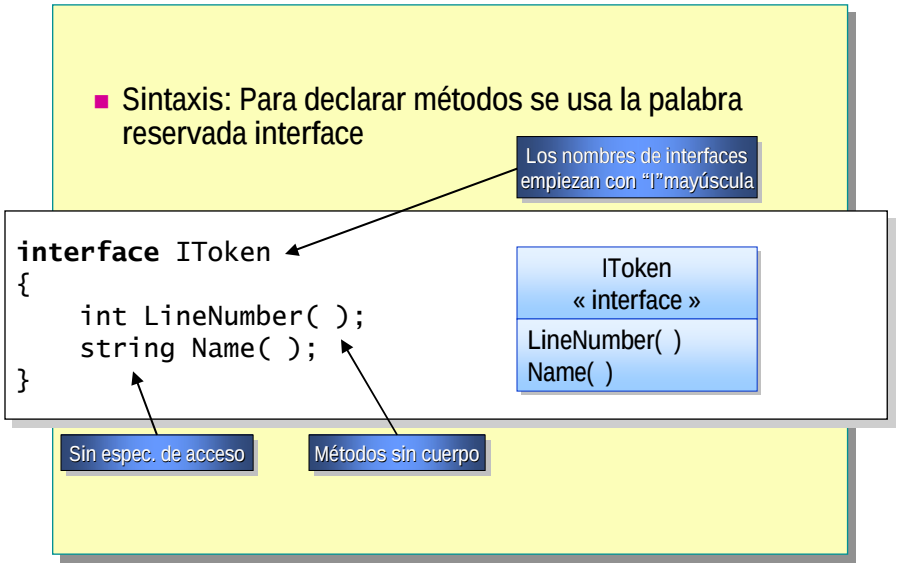
Al final de esta lección, usted será capaz de:

- Usar la sintaxis para declarar interfaces.
- Usar las dos técnicas de implementación de métodos de interfaz en clases derivadas.

Declaración de interfaces

Objetivo del tema
Explicar cómo declarar interfaces.

Explicación previa
Las interfaces son similares a las clases.



Para su información
No está permitido declarar tipos (como enums) dentro de una interfaz.

Una interfaz en C# se parece a una clase que no tiene código y se declara de forma similar, pero se usa la palabra reservada **interface** en vez de **class**. La sintaxis de esta palabra reservada está explicada en la transparencia.

Nota Se recomienda que todos los nombres de interfaces comiencen con una "I" mayúscula. Por ejemplo, **IToken** es mejor que **Token**.

Características de las interfaces

Las interfaces tienen las dos siguientes características importantes.

Los métodos de interfaz son implícitamente públicos

Los métodos que se declaran en una interfaz son públicos de forma implícita. Como consecuencia no se permite el uso de modificadores de acceso **public** explícitos, como se ve en el siguiente ejemplo:

```
interface IToken
{
    public int LineNumber ( ); // Error al compilar
}
```

Los métodos de interfaz no contienen cuerpo

Los métodos que se declaran en una interfaz no pueden contener cuerpo. Por ejemplo, el siguiente código no estaría permitido:

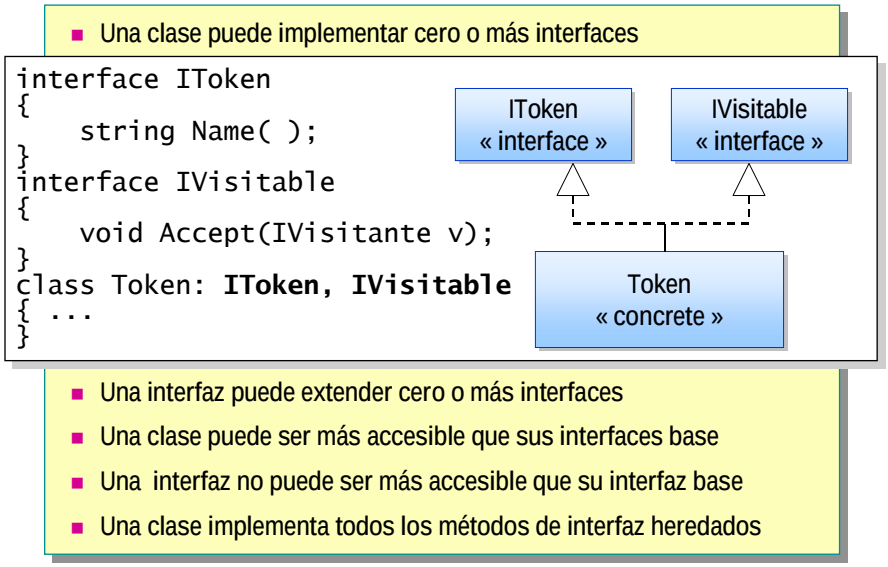
```
interface IToken
{
    int LineNumber ( ) { ... } // Error al compilar
}
```

Estrictamente hablando, las interfaces pueden contener declaraciones de propiedades de interfaz (que son declaraciones de propiedades sin cuerpo), declaraciones de eventos de interfaz (que son declaraciones de eventos sin cuerpo) y declaraciones de indizadores de interfaz (que son declaraciones de indizadores sin cuerpo).

Implementación de varias interfaces

Objetivo del tema
Explicar cómo se puede heredar de varias interfaces.

Explicación previa
Una clase implementa una interfaz.



Aunque C# permite únicamente la herencia sencilla, es posible implementar varias interfaces en una sola clase. Este tema discute las diferencias entre una clase y una interfaz con respecto a la implementación y extensión de interfaces, respectivamente, además de su accesibilidad en comparación con sus interfaces base.

Implementación de interfaces

Una clase puede implementar cero o más interfaces, pero no puede extender más de una clase de forma explícita. La transparencia muestra un ejemplo de este punto.

Nota Estrictamente hablando, una clase extiende siempre una clase. Si no se especifica la clase base, una clase heredará de **object** implícitamente.

Recomendación al profesor
Asegúrese de que los delegados usan la terminología correcta. Una clase extiende otra clase e implementa una interfaz.

En la transparencia se han dejado intencionadamente en blanco los métodos de Token. Las dos formas de implementar una interfaz se discutirán en las dos transparencias siguientes.

Un aspecto importante en este tema es que una clase debe implementar todos los métodos de interfaz heredados.

Probablemente también sea conveniente mencionar que una interfaz puede heredar de otras interfaces. Dé un ejemplo.

Por el contrario, una interfaz puede extender cero o más interfaces. Por ejemplo, el código de la transparencia se podría reescribir de la siguiente manera:

```
interface IToken { ... }
interface IVisitable { ... }
interface IVisitableToken: IVisitable, IToken { ... }
class Token: IVisitableToken { ... }
```

Accesibilidad

Una clase puede ser más accesible que sus interfaces base. Por ejemplo, es posible declarar una clase pública que implemente una interfaz privada, como se ve:

```
class Example
{
    private interface INested { }
    public class Anidada: INested { } // Okay
}
```

Sin embargo, una interfaz no puede ser más accesible que sus interfaces base. Es un error declarar una interfaz pública que extiende una interfaz privada, como se muestra en el siguiente ejemplo:

```
class Example
{
    private interface INested { }

    public interface IAlsoNested: INested { }
    // Error al compilar
}
```

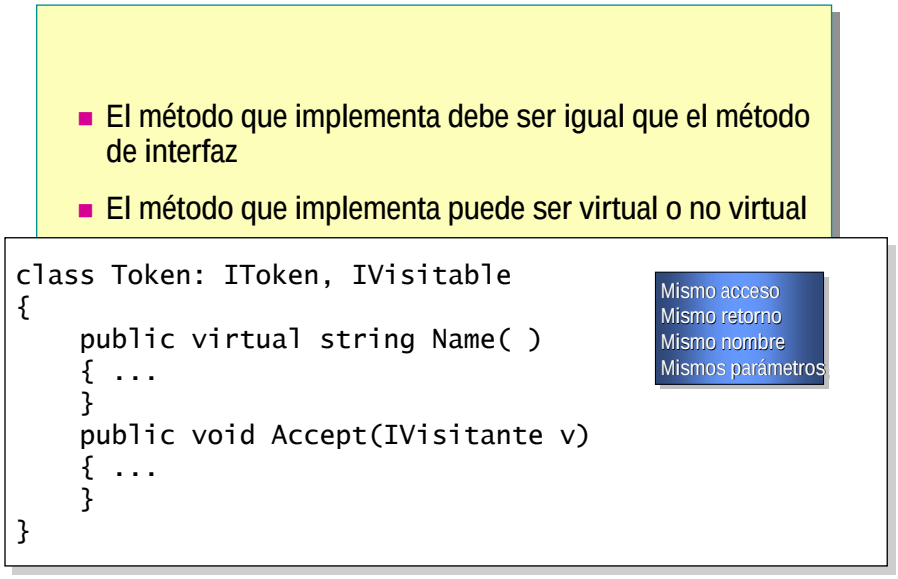
Métodos de interfaz

Una clase debe implementar todos los métodos de todas las interfaces que extienda, independientemente de que las interfaces se hereden directa o indirectamente.

Implementación de métodos de interfaz

Objetivo del tema
Discutir la implementación de interfaces.

Explicación previa
Hay una serie de reglas que se deben cumplir cuando se implementa una interfaz en una clase.



Si una clase implementa una interfaz, tiene que implementar todos los métodos declarados en esa interfaz. Este requisito es muy conveniente, ya que las interfaces no pueden definir los cuerpos de sus propios métodos.

El método que la clase implementa debe ser absolutamente idéntico al método de la interfaz. Tiene que ser igual en:

- Acceso

  Un método de interfaz es público de forma implícita, lo que significa que el método que lo implementa debe estar declarado explícitamente como público. Si se omitiera el modificador de acceso, el método sería privado por defecto.

- Tipo de retorno

  Si el tipo de retorno en la interfaz está declarado como **T**, el tipo de retorno en la clase que la implementa no puede estar declarado como un tipo derivado de **T**, sino que debe ser **T**. En otras palabras, C# no permite la covarianza de tipos de retorno.

- Nombre

  No hay que olvidar que C# distingue mayúsculas y minúsculas en los nombres.

- Lista de parámetros

El siguiente código cumple todos estos requisitos:

```
interface IToken
{
    string Name( );
}
interface IVisitable
{
    void Accept(IVisitante v);
}
class Token: IToken, IVisitable
{
    public virtual string Name( )
    { ...
    }
    public void Accept(IVisitante v)
    { ...
    }
}
```

El método que implementa puede ser virtual, como **Name** en el código anterior. En ese caso, es posible sustituir el método en otras clases derivadas. Por otro lado, el método que implementa también puede ser no virtual, como **Accept** en el mismo código. En este último caso, no es posible sustituir el método en otras clases derivadas.

Uso de clases abstractas

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En esta lección aprenderemos a usar clases abstractas.

- Declaración de clases abstractas
- Uso de clases abstractas en una jerarquía de clases
- Comparación de clases abstractas e interfaces
- Implementación de métodos abstractos
- Uso de métodos abstractos

Las clases abstractas se emplean para proporcionar implementaciones parciales de clases que se completan con clases derivadas concretas. Las clases abstractas son especialmente útiles para la implementación parcial de una interfaz que puede ser reutilizada por varias clases derivadas.

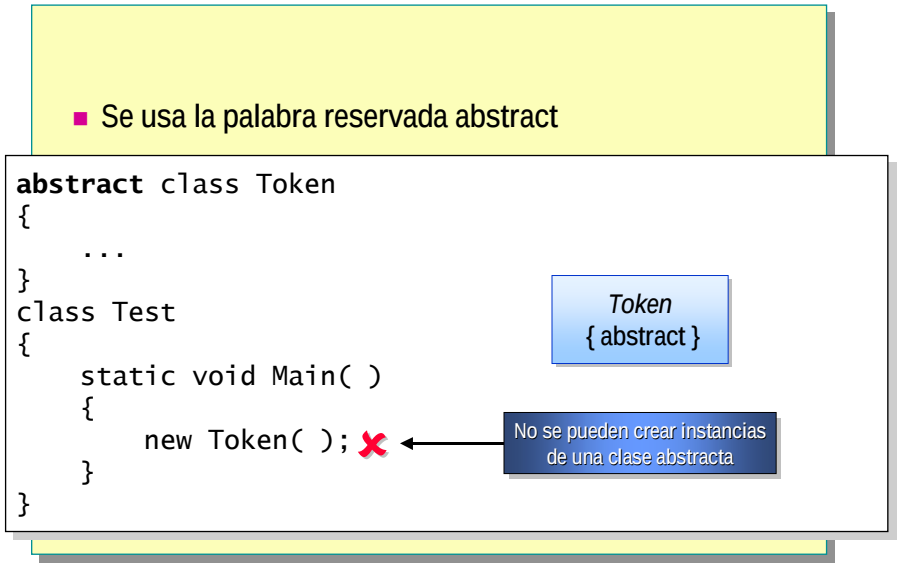
Al final de esta lección, usted será capaz de:

- Usar la sintaxis para la declaración de una clase abstracta.
- Explicar el uso de clases abstractas en una jerarquía de clases.

Declaración de clases abstractas

Objetivo del tema
Describir la sintaxis para declarar una clase abstracta.

Explicación previa
Una clase abstracta es una clase de la que no se pueden crear instancias.



Para declarar una clase abstracta se utiliza la palabra reservada **abstract**, como se muestra en la transparencia.

Las reglas que rigen el uso de una clase abstracta son prácticamente las mismas que se aplican a una clase no abstracta. Las únicas diferencias entre usar clases abstractas y no abstractas son:

- No está permitido crear una instancia de una clase abstracta.

  En este sentido, las clases abstractas son como interfaces.
- Se puede crear un método abstracto en una clase abstracta.

  Se puede declarar un método abstracto en una clase abstracta, pero no en una que no lo sea.

Las clases abstractas comparten las siguientes características con las clases no abstractas:

- Extensibilidad limitada

  Una clase abstracta puede extender como máximo otra clase o clase abstracta. Obsérvese que una clase abstracta puede extender una clase que no sea abstracta
- Múltiples interfaces

  Una clase abstracta puede implementar varias interfaces
- Métodos de interfaz heredados
- Una clase abstracta debe implementar todos los métodos de interfaz heredados

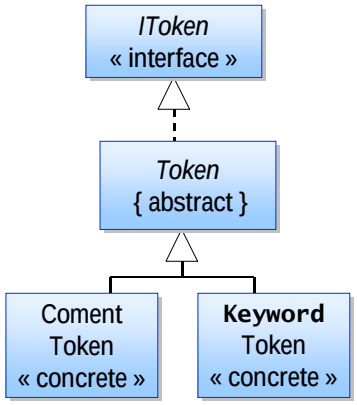
Uso de clases abstractas en una jerarquía de clases

Objetivo del tema
Ilustrar el uso de clases abstractas en la implementación de interfaces.

Explicación previa
Las clases abstractas se emplean a menudo para la implementación parcial de interfaces.

■ Ejemplo 1

```
interface IToken
{
    string Name( );
}

abstract class Token: IToken
{
    string IToken.Name( )
    {   ...
    }
    ...
}

class ComentToken: Token
{   ...
}
class KeywordToken: Token
{   ...
}
```



La función de las clases abstractas en una jerarquía clásica de tres niveles, formada por una interfaz, una clase abstracta y una clase concreta, es proporcionar una implementación completa o parcial de una interfaz.

Una clase abstracta que implementa una interfaz

Consideremos el Ejemplo 1 en la transparencia, en el que la clase abstracta implementa una interfaz. Es una implementación explícita del método de interfaz. La implementación explícita no es virtual y por tanto no se puede sustituir en las demás clases derivadas, como **ComentToken**.

Sin embargo, **CommentToken** puede reimplementar la interfaz **IToken** de la siguiente manera:

```
interface IToken
{
    string Name( );
}

abstract class Token: IToken
{
    string IToken.Name( ) { ... }
}

class ComentToken: Token, IToken
{
    public virtual string Name( ) { ... }
}
```

Como se ve, en este caso no es necesario marcar **ComentToken**.**Name** como método **new**. Esto se debe a que una clase derivada sólo puede ocultar un método de clase base visible, pero la implementación explícita de **Name** en **Token** no es directamente visible en **ComentToken**.

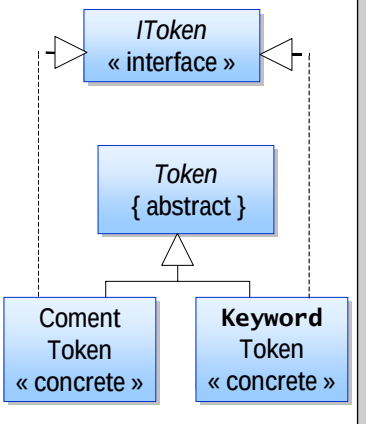
Uso de clases abstractas en una jerarquía de clases *(cont.)*

Objetivo del tema
Ilustrar el uso de clases abstractas en la implementación de interfaces.

Explicación previa
Aquí se da otro ejemplo del uso de clases abstractas para la implementación parcial de interfaces.

- Ejemplo 2

```
interface IToken
{
    string Name( );
}

abstract class Token
{
    public virtual string Name( )
    {   ...
    }
    ...
}

class ComentToken: Token, IToken
{   ...
}
class KeywordToken: Token, IToken
{   ...
}
```



Esta transparencia muestra otro ejemplo para continuar la discusión de la función de las clases abstractas en una jerarquía clásica de tres niveles.

Una clase abstracta que no implementa una interfaz

Recomendación al profesor
La discusión del Ejemplo 2 tiene dos objetivos principales:
1. Mostrar la sintaxis que se utiliza para extender una clase e implementar una o más interfaces.
2. Explicar que una clase derivada puede heredar su implementación de una interfaz desde la clase base que extiende, incluso en caso de que esa clase base no implemente esa interfaz.

Consideremos el Ejemplo 2 en la transparencia, en el que la clase abstracta no implementa la interfaz. Esto significa que la única forma de que proporcione una implementación de la interfaz a otra clase derivada concreta es mediante un método público. Opcionalmente, el método puede estar declarado como virtual en la clase abstracta para que sea posible sustituirlo en las clases, como se muestra en el siguiente código:

```
interface IToken
{
    string Name( );
}
abstract class Token
{
    public virtual string Name( ) { ... }
}
class ComentToken: Token, IToken
{
    public override string Name( ) { ... } // Okay
}
```

Como se ve, una clase puede heredar su interfaz y su implementación de esa interfaz desde distintas ramas de la herencia.

Comparación de clases abstractas e interfaces

Objetivo del tema
Comparar clases abstractas e interfaces.

Explicación previa
Entre las clases abstractas y las interfaces existen algunos parecidos, pero también muchas diferencias.

- Parecidos
  - No se pueden crear instancias de ninguna de ellas
  - No se puede sellar ninguna de ellas
- Diferencias
  - Las interfaces no pueden contener implementaciones
  - Las interfaces no pueden declarar miembros no públicos
  - Las interfaces no pueden extender nada que no sea una interfaz

Tanto las clases abstractas como las interfaces existen para derivar otras clases de ellas (o ser implementadas). Sin embargo, una clase puede extender como máximo una clase abstracta, por lo que hay que tener más cuidado cuando se deriva de una clase abstracta que cuando se deriva de una interfaz. Las clases abstractas se deben utilizar solamente para implementar relaciones del tipo “es un”.

Los parecidos entre clases abstractas e interfaces son:

- No se pueden crear instancias de ellas.

  Esto significa que no es posible usarlas directamente para crear objetos.
- No se pueden sellar.

  Esto es normal, ya que no es posible implementar una interfaz sellada.

La siguiente tabla resume las diferencias entre clases abstractas e interfaces.

Interfaces	Clases abstractas
No pueden contener implementación	Pueden contener implementación
No pueden declarar miembros no públicos	Pueden declarar miembros no públicos
Pueden extender sólo otras interfaces	Pueden extender otras clases, que pueden no ser abstractas

Cuando se comparan clases abstractas e interfaces, se puede pensar en las clases abstractas como clases sin terminar que contienen los planes para lo que falta por hacer.

Implementación de métodos abstractos

Objetivo del tema
Explicar cómo se implementan métodos abstractos.

Explicación previa
Una clase derivada puede sustituir un método virtual, pero debe sustituir un método abstracto.

- Sintaxis: Se usa la palabra reservada abstract

```
abstract class Token
{
    public virtual string Name( ) { ... }
    public abstract int Longitud( );
}
class ComentToken: Token
{
    public override string Name( ) { ... }
    public override int Longitud( ) { ... }
}
```

- Los métodos abstractos no pueden tener cuerpo

Para declarar un método abstracto hay que añadir el modificador **abstract** a la declaración del método. La sintaxis del modificador **abstract** se muestra en la transparencia.

Sólo clases abstractas pueden declarar métodos abstractos, como se ve en este ejemplo:

```
interface IToken
{
    abstract string Name( ); // Error al compilar
}
class CommentToken
{
    abstract string Name( ); // Error al compilar
}
```

Recomendación al profesor
Es obligatorio sustituir un método abstracto, pero no es necesario sustituir un método virtual.

Nota Los desarrolladores en C++ pueden considerar los métodos abstractos como equivalentes a los métodos virtuales puros de C++.

Los métodos abstractos no pueden tener cuerpo

Los métodos abstractos no pueden contener ninguna implementación, como queda claro en el siguiente código:

```
abstract class Token
{
    public abstract string Name( ) { ... }
    // Error al compilar
}
```

Uso de métodos abstractos

Objetivo del tema
Describir las restricciones de los métodos abstractos.

Explicación previa
Antes de implementar un método abstracto hay que tener en cuenta sus características y las restricciones que impone.

- Los métodos abstractos son virtuales
- Los métodos override pueden sustituir a métodos abstractos en otras clases derivadas
- Los métodos abstractos pueden sustituir a métodos de la clase base declarados como virtuales
- Los métodos abstractos pueden sustituir a métodos de la clase base declarados como override

Al implementar métodos abstractos hay que tener en cuenta lo siguiente:

- Los métodos abstractos son virtuales.
- Los métodos override pueden sustituir a métodos abstractos en otras clases derivadas.
- Los métodos abstractos pueden sustituir a métodos de la clase base declarados como virtuales.
- Los métodos abstractos pueden sustituir a métodos de la clase base declarados como override.

A continuación se describe en detalle cada uno de estos puntos.

Los métodos abstractos son virtuales

Los métodos abstractos se consideran implícitamente virtuales pero no pueden estar marcados como virtuales de forma explícita, como se ve en el siguiente código:

```
abstract class Token
{
    public virtual abstract string Name( ) { ... }
    // Error al compilar
}
```

Los métodos override pueden sustituir a métodos abstractos en otras clases derivadas

Al ser implícitamente virtuales, es posible sustituir métodos abstractos en clases derivadas. A continuación se muestra un ejemplo:

```
class ComentToken: Token
{
    public override string Name( ) {...}
}
```

Recomendación al profesor

Puede utilizar el siguiente fragmento para ver hasta qué punto los estudiantes comprenden este aspecto:

```
abstract class A
{
    public
abstract
    void M( );
}
abstract class B:
A
{
    public
abstract
    void M( );
}
```

Esto no está permitido. Una clase no podría implementar A.M y B.M.

Los métodos abstractos pueden sustituir a métodos de la clase base declarados como virtuales

La sustitución de un método de clase base declarado como virtual fuerza a otra clase derivada a tener su propia implementación del método, y hace que no se pueda utilizar la implementación original del método. A continuación se muestra un ejemplo:

```
class Token
{
    public virtual string Name( ) { ... }
}
abstract class Force: Token
{
    public abstract override string Name( );
}
```

Los métodos abstractos pueden sustituir a métodos de la clase base declarados como override

La sustitución de un método de clase base declarado como override fuerza a otra clase derivada a tener su propia implementación del método, y hace que no se pueda utilizar la implementación original del método. A continuación se muestra un ejemplo:

```
class Token
{
    public virtual string Name( ) { ... }
}
class AnotherToken: Token
{
    public override string Name( ) { ... }
}
abstract class Force: AnotherToken
{
    public abstract override string Name( );
}
```