Programación con C# .NET

Tema 3:
Orientación a Objetos con C#

El lenguaje C# y la orientación a objetos

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

“Hola, mundo” de nuevo

```
using System;

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

Microsoft C#.net

Definición de clases simples

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

```
class CuentaBancaria
{
    public void Retirar(decimal cantidad)
    { ... }
    public void Ingresar(decimal cantidad)
    { ... }
    private decimal balance;
    private string name;
}
```

Métodos públicos describen un comportamiento accesible

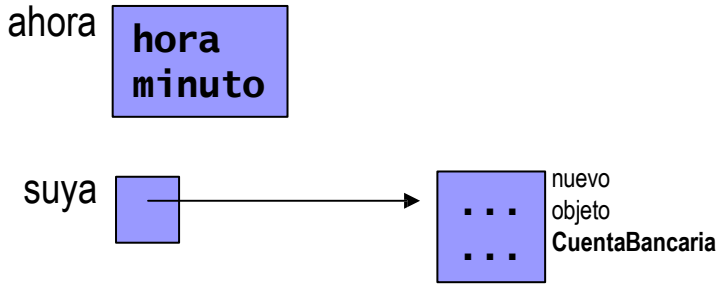
Campos privados describen un estado inaccesible

Microsoft C#.net

Instancias de nuevos objetos

- 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);
    }
}
```



Microsoft C#.net

Uso de la palabra reservada this

- 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;
}
```

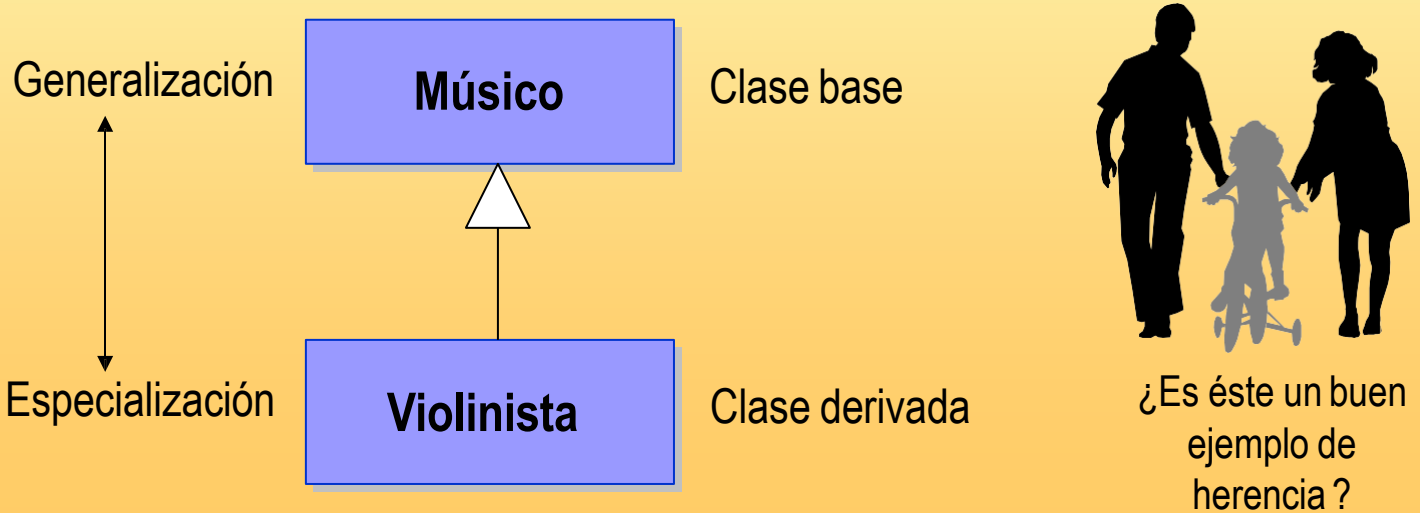
Si esta instrucción fuera
nombre = nombre;
¿qué ocurriría?

Definición de sistemas orientados a objetos

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

Herencia

- La herencia indica una relación “es un tipo de"
  - La herencia es una relación entre clases
  - Las nuevas clases añaden especialización a las existentes

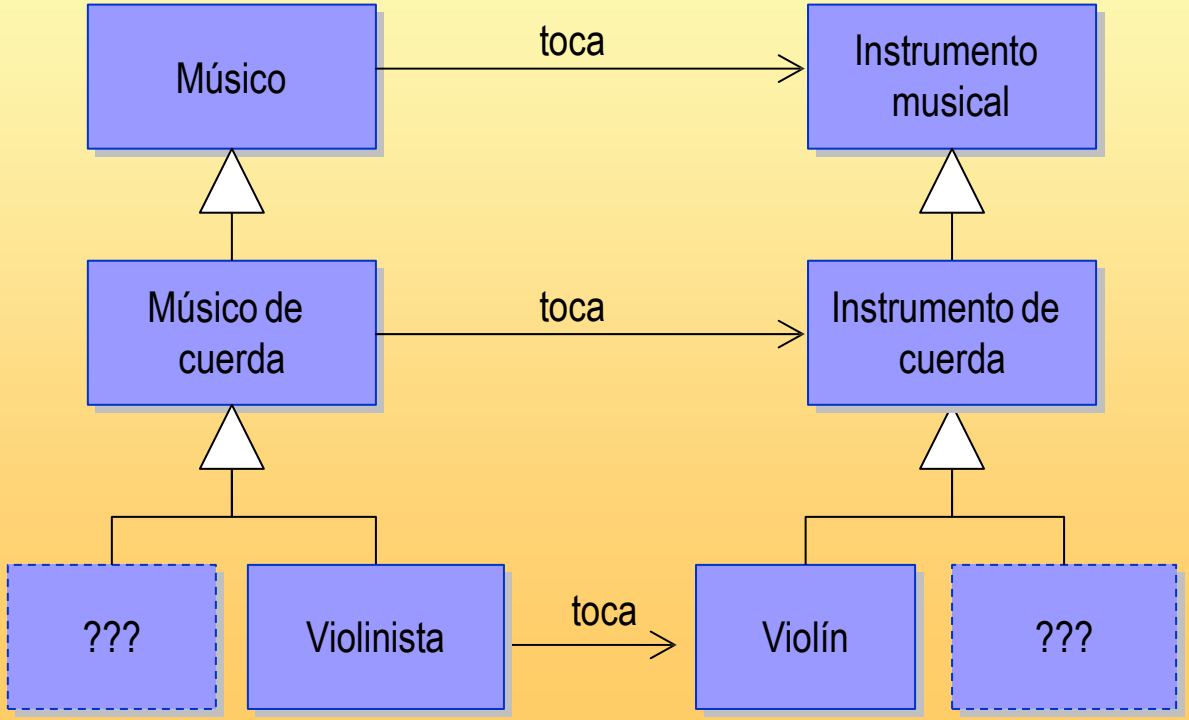


¿Es éste un buen ejemplo de herencia ?

Microsoft C#.net

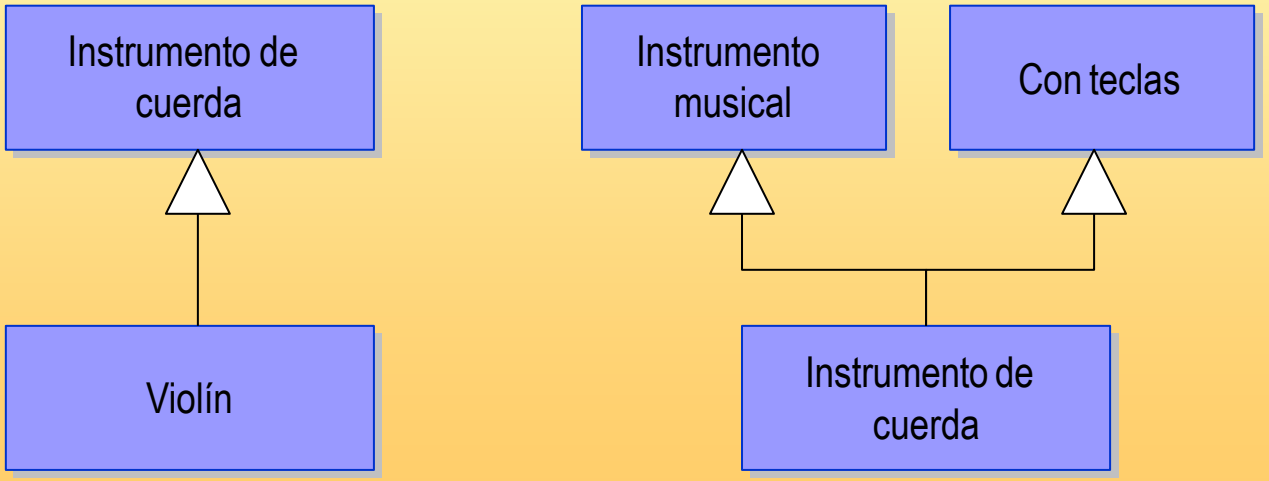
Jerarquías de clases

- Las clases con relaciones de herencia forman jerarquías de clases



Herencia sencilla y múltiple

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

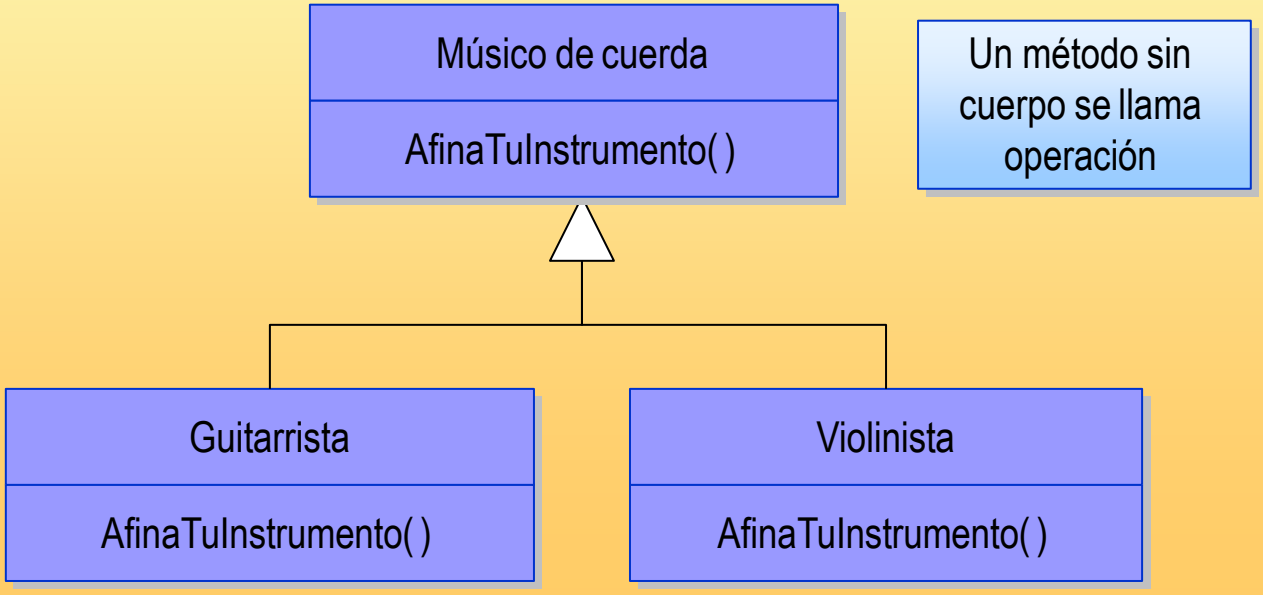


Violín tiene una sola clase base directa

Instrumento de cuerda tiene dos clases base directas

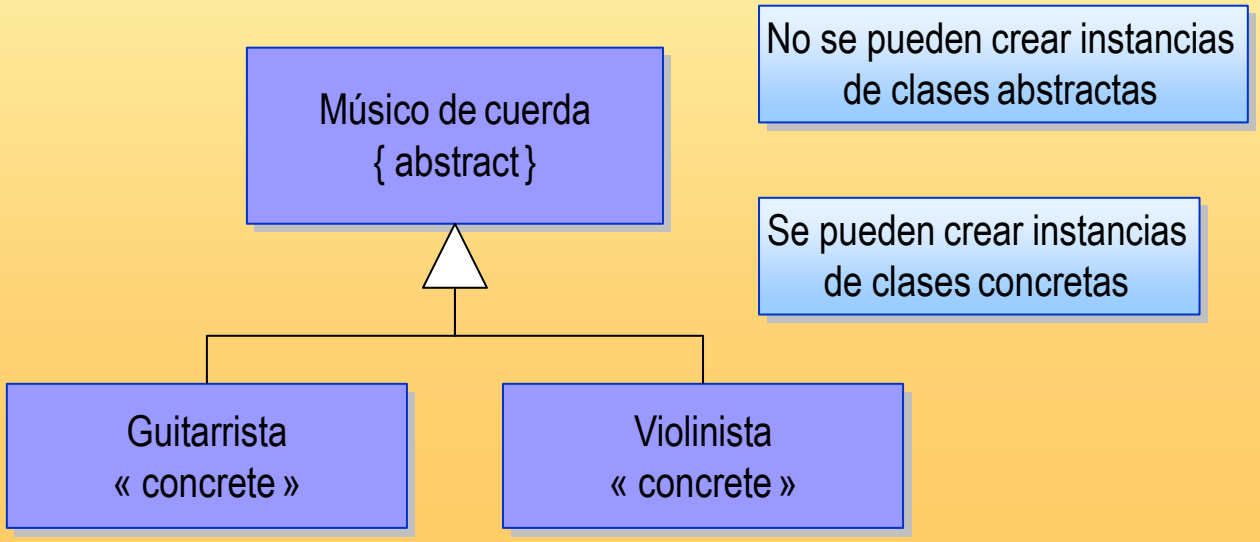
Polimorfismo

- El nombre del método reside en la clase base
- Los distintos cuerpos del método residen en las clases derivadas



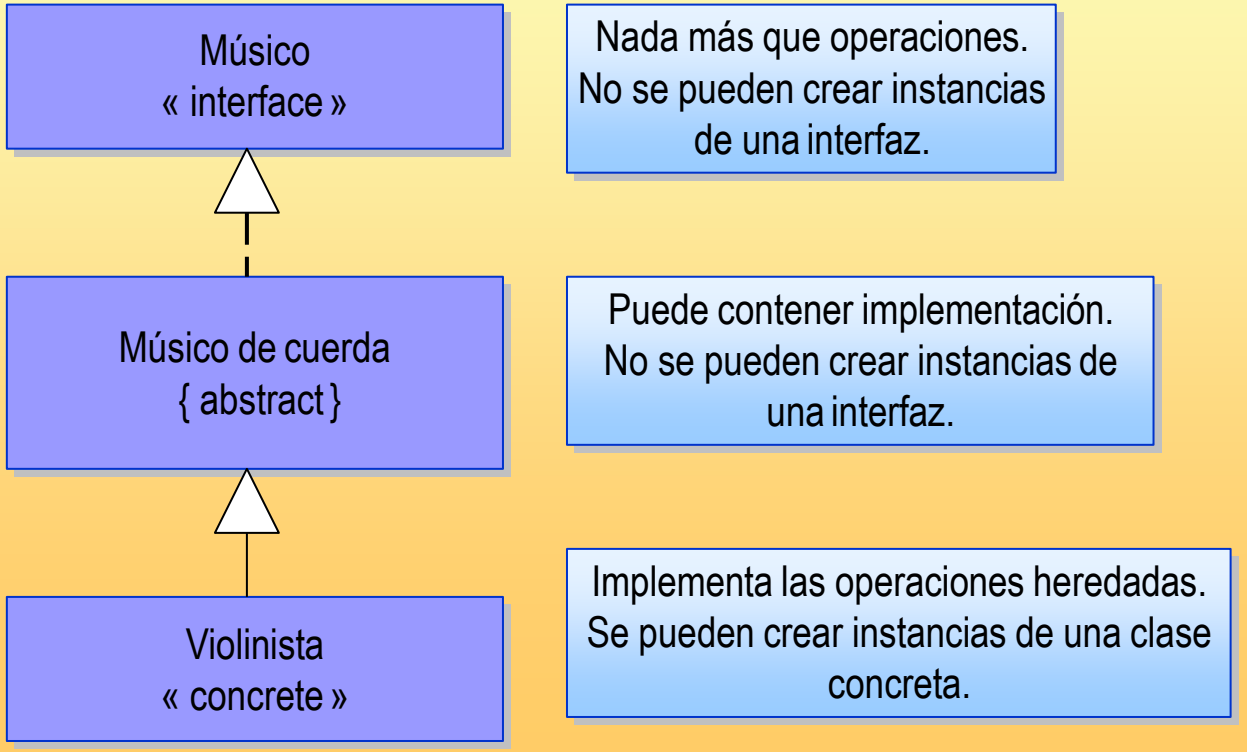
Clases base abstractas

- Algunas clases existen sólo para ser clases base
  - No tiene sentido crear instancias de estas clases
  - Estas clases son *abstractas*



Interfaces

- Las interfaces contienen sólo operaciones, no implementación



Uso de variables de tipo referencia

Microsoft C#.net

Notas generales

- Uso de variables de tipo referencia
- Uso de tipos referencia comunes
- La jerarquía de objetos
- Espacios de nombres en .NET Framework

Uso de variables de tipo referencia

- Comparación de tipos valor y tipos referencia
- Declaración y liberación de variables referencia
- Referencias no válidas
- Comparación de valores y comparación de referencias
- Referencias múltiples a un mismo objeto
- Uso de referencias como parámetros de métodos

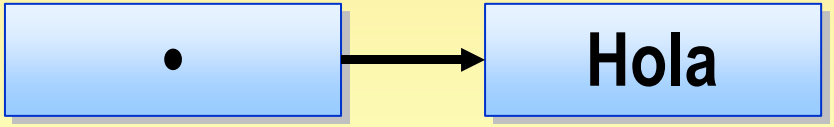
Comparación de tipos valor y tipos referencia

- **Tipos valor**
  - La variable contiene el valor directamente
  - Ejemplos: **char**, **int**

```
int mol;
mol = 42;
```

- **Tipos referencia**
  - La variable contiene una referencia al dato
  - El dato se almacena en un área de memoria aparte

```
string mol;
mol = "Hola";
```

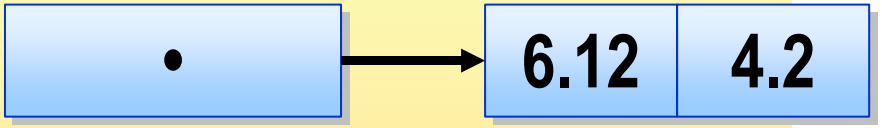


Microsoft C#.net

Declaración y liberación de variables referencia

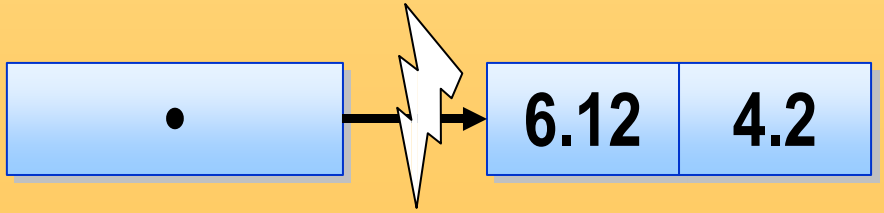
- Declaración de variables referencia

```
coordenada c1;
c1 = new coordinate();
c1.x = 6.12;
c1.y = 4.2;
```



- **Liberación de variables referencia**

```
c1 = null;
```



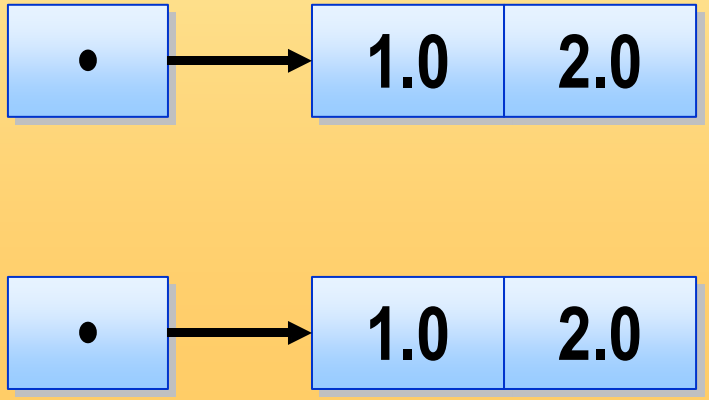
Microsoft C#.net

Referencias no válidas

- Si hay referencias no válidas
  - No es posible acceder a miembros o variables
- Referencias no válidas en tiempo de compilación
  - El compilador detecta el uso de referencias no inicializadas
- Referencias no válidas en tiempo de ejecución
  - El sistema generará un error de excepción

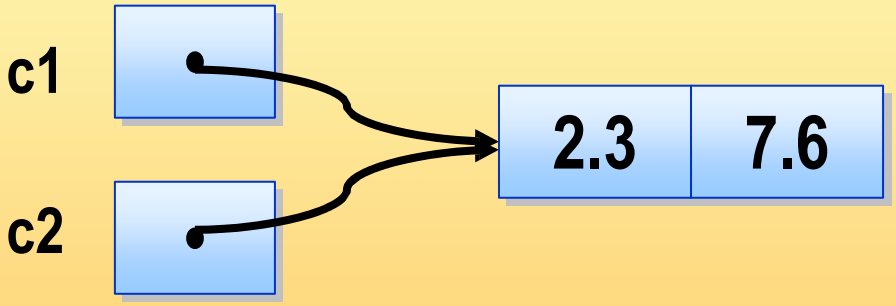
Comparación de valores y comparación de referencias

- Comparación de tipos valor
  - == and != comparan valores
- Comparación de tipos referencia
  - == and != comparan las referencias, no los valores



Referencias múltiples a un mismo objeto

- Dos referencias pueden apuntar a un mismo objeto
  - Dos formas de acceder a un mismo objeto para lectura/escritura

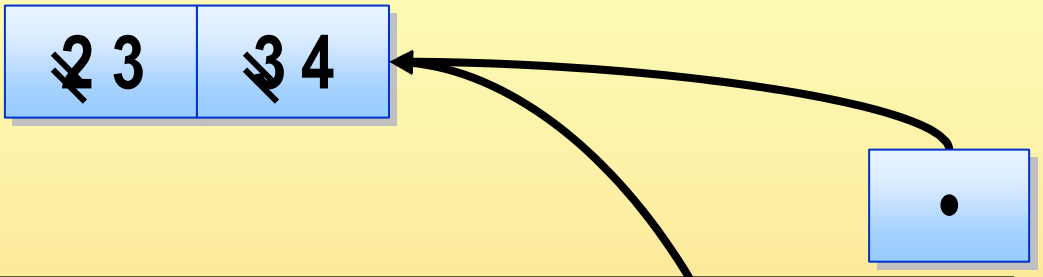



```
coordinate c1= new coordinate ( );
coordinate c2;
c1.x = 2.3; c1.y = 7.6;
c2 = c1;
Console.WriteLine(c1.x + " , " + c1.y);
Console.WriteLine(c2.x + " , " + c2.y);
```

Microsoft C#.net

Uso de referencias como parámetros de métodos

- Las referencias se pueden usar como parámetros
  - Si se pasan por valor, es posible cambiar el dato referenciado




```
static void PassCoordinateByValue(coordinate c)
{
  c.x++; c.y++;
}
```

```
loc.x = 2; loc.y = 3;
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Microsoft C#.net

Uso de tipos referencia comunes

- Clase Exception
- Clase String
- Métodos, operadores y propiedades comunes de String
- Comparaciones de cadenas de caracteres
- Operadores de comparación en String

Clase Exception

- Exception es una clase
- Los objetos Exception se usan para lanzar excepciones
  - Para crear un objeto **Exception** se usa **new**
  - Para lanzar el objeto se usa **throw**
- Los tipos Exception son subclases de Exception

Clase String

- Cadenas de caracteres Unicode
- Abreviatura de System.String
- Inmutable

```
string s = "Hola";

s[0] = 'c'; // Error al compilar
```

Métodos, operadores y propiedades comunes de String

- Corchetes
- Método Insert
- Propiedad Length
- Método Copy
- Método Concat
- Método Trim
- Métodos ToUpper y ToLower

Comparaciones de cadenas de caracteres

- Método Equals
  - Comparación de valores
- Método Compare
  - Más comparaciones
  - Opción para no distinguir mayúsculas y minúsculas
  - Ordenación alfabética
- Opciones locales de Compare

Operadores de comparación en String

- Los operadores == y != están sobrecargados para cadenas
- Son equivalentes a String.Equals y !String.Equals

```
string a = "Test";
string b = "Test";
if (a == b) ...   // Devuelve true
```

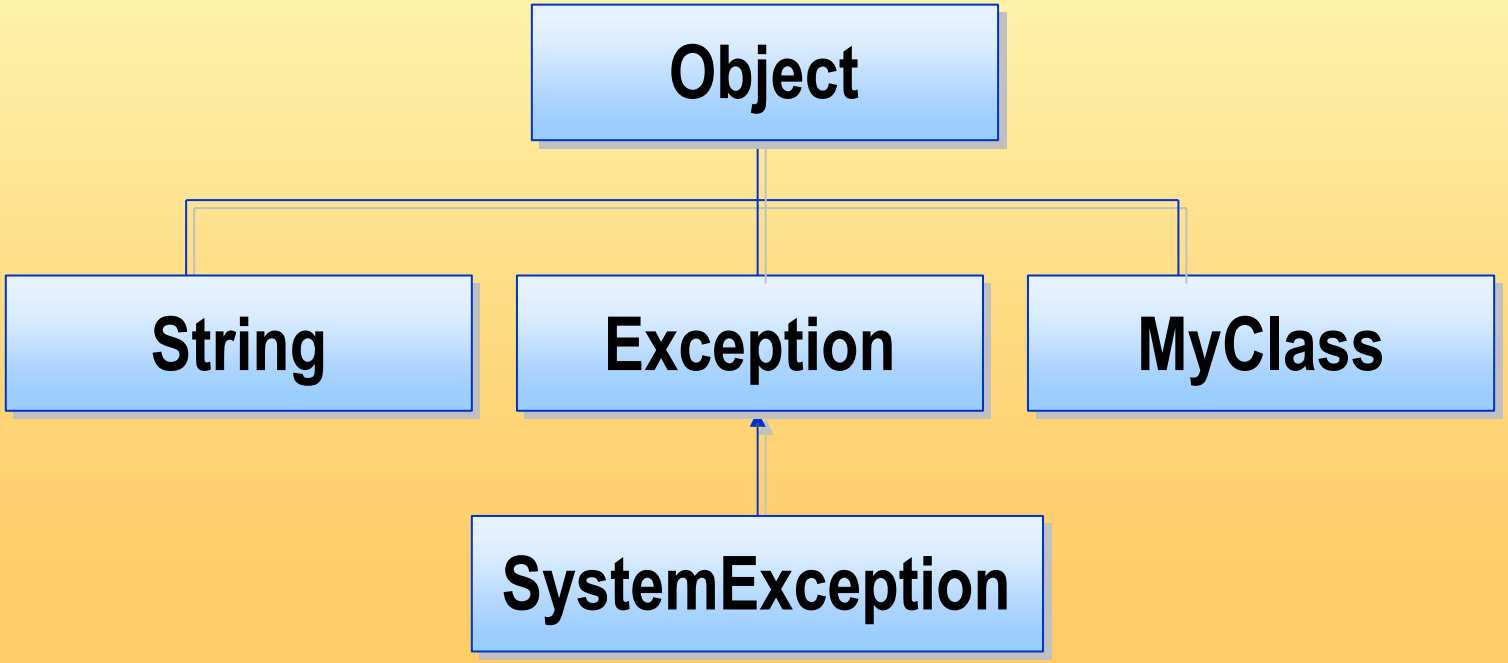
La jerarquía de objetos

- El tipo object
- Métodos comunes
- Reflexión

Microsoft C#.net

El tipo object

- Sinónimo de System.Object
- Clase base para todas las demás clases



Métodos comunes

- Métodos comunes para todos los tipos referencia
  - Método **ToString**
  - Método **Equals**
  - Método **GetType**
  - Método **Finalize**

Reflexión

- Es posible obtener información sobre el tipo de un objeto
- Espacio de nombres System.Reflection
- El operador typeof devuelve el tipo de un objeto
  - Sólo para clases en tiempo de compilación
- Método GetType en System.Object
  - Información sobre clases en tiempo de ejecución

Conversiones de datos

- Conversión de tipos valor
- Conversiones padre/hija
- El operador is
- El operador as
- Conversiones y el tipo object
- Conversiones e interfaces
- Boxing y unboxing

Conversión de tipos valor

- Conversiones implícitas
- Conversiones explícitas
  - Operador de cast
- Excepciones
- Clase System.Convert
  - Control interno de conversiones

Microsoft C#.net

Conversiones padre/hija

- Conversión a referencia de clase padre
  - Implícita o explícita
  - Siempre tiene éxito
  - Siempre se puede asignar a un objeto
- Conversión a referencia de clase hija
  - Es necesario cast explícito
  - Comprueba que la referencia es del tipo correcto
  - Si no lo es, causa una excepción **InvalidCastException**

El operador is

- Devuelve true si es posible realizar una conversión

```
Pajaro b;
if (a is Pajaro)
    b = (Pajaro) a; // No hay problema
else
    Console.WriteLine("No es Pájaro");
```

El operador as

- Hace conversiones entre tipos referencia, como cast
- En caso de error
  - Devuelve null
  - No causa una excepción

```
Pajaro b = a as Pajaro; // Convertir

if (b == null)
      Console.WriteLine("No es Pájaro”);
```

Conversiones y el tipo object

- El tipo object es la base para todas las clases
- Se puede asignar a object cualquier referencia
- Se puede asignar cualquier variable object a cualquier referencia
  - Con conversión de tipo y comprobaciones
- El tipo object y el operador is

```
object buey;
buey = a;
buey = (object) a;
buey = a as object;
```

```
b = (Pajaro) buey;
b = buey as Pajaro;
```

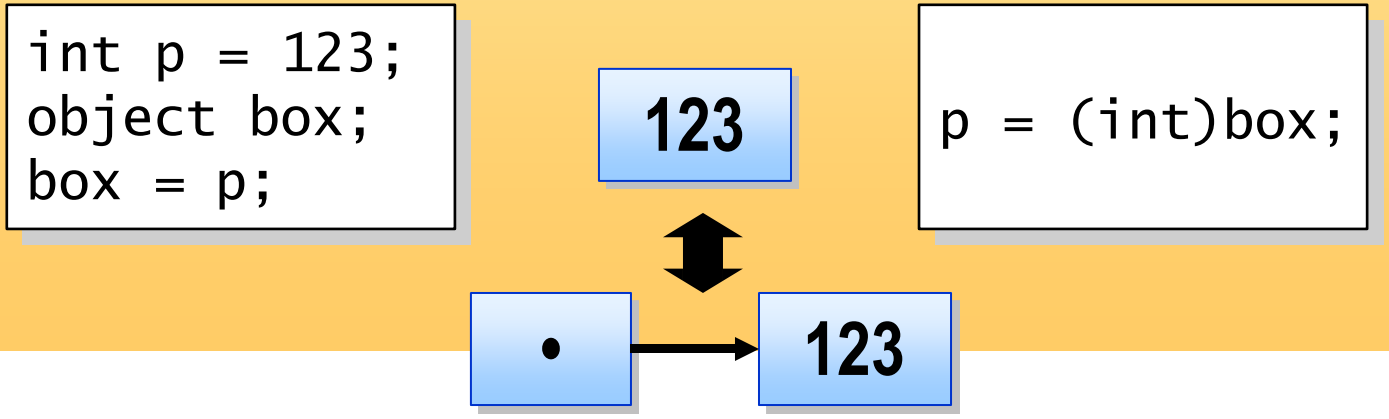
Microsoft C#.net

Conversiones e interfaces

- Una interfaz sólo se puede usar para acceder a sus propios miembros
- No es posible acceder a otros miembros y variables de la clase a través de la interfaz

Boxing y unboxing

- Sistema de tipos unificado
- Boxing
- Unboxing
- Llamadas a métodos de object en tipos valor



Microsoft C#.net

Creación y destrucción de objetos

Microsoft C#.net

Descripción general

- Uso de constructores
- Objetos y memoria

Uso de constructores

- Creación de objetos
- Uso del constructor por defecto
- Sustitución del constructor por defecto
- Sobrecarga de constructores

Creación de objetos

- Paso 1: Asignación de memoria
  - Se usa **new** para asignar memoria desde el montón
- Paso 2: Inicialización del objeto usando un constructor
  - Se usa el nombre de la clase seguido por paréntesis

```
Fecha cuando = new Date( );
```

Uso del constructor por defecto

- Características de un constructor por defecto
  - Acceso público
  - Mismo nombre que la clase
  - No tiene tipo de retorno (ni siquiera **void**)
  - No recibe ningún argumento
  - Inicializa todos los campos a **cero**, **false** o **null**
- Sintaxis del constructor

```
class Date { public Date( ) { ... } }
```

Sustitución del constructor por defecto

- El constructor por defecto puede no ser adecuado
  - En ese caso no hay que usarlo, sino escribir otro

```
Class Date
{
    public Date( )
    {
        ssaa = 1970;
        mm = 1;
        dd = 1;
    }
    private int ccyy, mm, dd;
}
```

Microsoft C#.net

Sobrecarga de constructores

- Los constructores son métodos y pueden estar sobrecargados
  - Mismo ámbito, mismo nombre, distintos parámetros
  - Permite inicializar objetos de distintas maneras
- AVISO
  - Si se escribe un constructor para una clase, el compilador no creará un constructor por defecto

```
Class Date
{
    public Date( ) { ... }
    public Date(int anno, int mes, int dia) { ... }
    ...
}
```

Microsoft C#.net

Objetos y memoria

- Tiempo de vida de un objeto
- Objetos y ámbito
- Recolección de basura

Microsoft C#.net

Tiempo de vida de un objeto

- Creación de objetos
  - Se usa **new** para asignar memoria
  - Se usa un constructor para inicializar un objeto en esa memoria
- Uso de objetos
  - Llamadas a métodos
- Destrucción de objetos
  - Se vuelve a convertir el objeto en memoria
  - Se libera la memoria

Objetos y ámbito

- El tiempo de vida de un valor a local está vinculado al ámbito en el que está declarado
  - Tiempo de vida corto (en general)
  - Creación y destrucción deterministas
- El tiempo de vida de un objeto dinámico no está vinculado a su ámbito
  - Tiempo de vida más largo
  - Destrucción no determinista

Recolección de basura

- No es posible destruir objetos de forma explícita
  - C# no incluye un inverso de **new** (como **delete**)
  - Ello se debe a que una función de eliminación explícita es una importante fuente de errores en otros lenguajes
- Los objetos se destruyen por recolección de basura
  - Busca objetos inalcanzables y los destruye
  - Los convierte de nuevo en memoria binaria no utilizada
  - Normalmente lo hace cuando empieza a faltar memoria

Herencia en C#

Microsoft C#.net

Notas generales

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

Microsoft C#.net

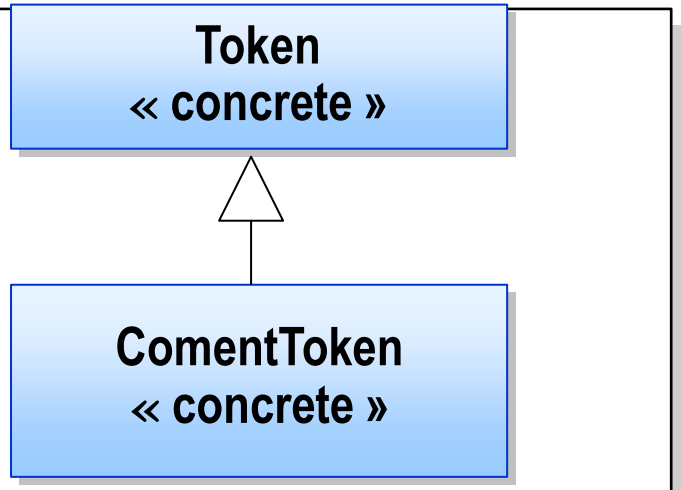
Derivación de clases

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

Extensión de clases base

- Sintaxis para derivar una clase desde una clase base

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



- Una clase derivada hereda la mayor parte de los elementos de su clase base
- Una clase derivada no puede ser más accesible que su clase base

Acceso a miembros de la clase

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

```
class Outside
{
     void Fails(Token t)
     {
          ...
        t.name      3
          ...
     }
}
```

- 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

Llamadas a constructores de la clase base

- Las declaraciones de constructores deben usar la palabra base

```
class Token
{
   protected Token(string name) { ... }
   ...
}
class ComentToken: Token
{
   public ComentToken(string name) : base(name) { }
   ...
}
```

- Una clase derivada no puede acceder a un constructor privado de la clase base
- Se usa la palabra base para habilitar el ámbito del identificador

Microsoft C#.net

Implementación de métodos

- 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

Definición de métodos virtuales

- Sintaxis: Se declara como virtual

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

- Los métodos virtuales son polimórficos

Uso de métodos virtuales

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

Sustitución de métodos (override)

- Sintaxis: Se usa la palabra reservada override

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

Uso de métodos override

- 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( ) { ... }
}
```

3

- 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

Microsoft C#.net

Uso de new para ocultar métodos

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

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

}
```

Uso de la palabra reservada new

- Ocultar tanto métodos virtuales como no virtuales

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

- Resolver conflictos de nombre en el código
- Ocultar métodos que tengan signaturas idénticas

Uso de clases selladas

- 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 { ... }
}
```

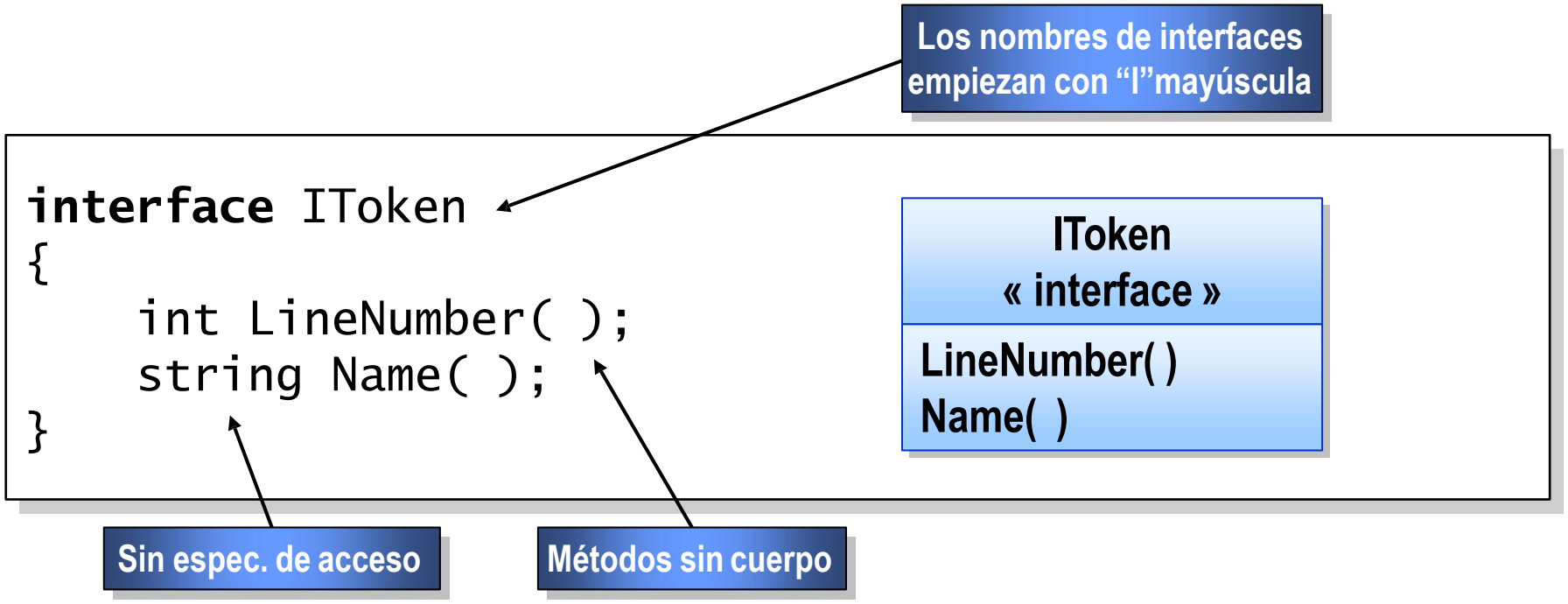
[3]

- # Uso de interfaces

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

Declaración de interfaces

- Sintaxis: Para declarar métodos se usa la palabra reservada interface

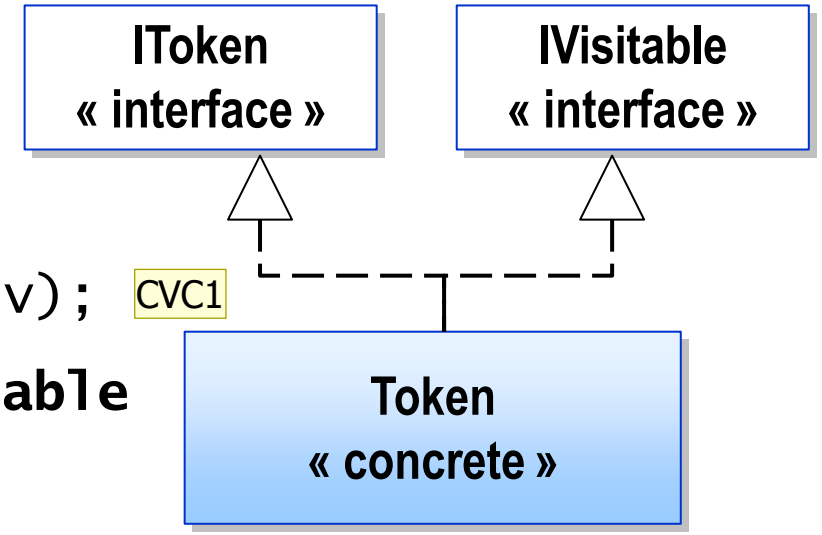


Microsoft C#.net

Implementación de varias interfaces

- Una clase puede implementar cero o más interfaces

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



- Una clase puede ser más accesible que sus interfaces base
- Una interfaz no puede ser más accesible que su interfaz base
- Una clase implementa todos los métodos de interfaz heredados

Implementación de métodos de interfaz

- El método que implementa debe ser igual que el método de interfaz
- El método que implementa puede ser virtual o no virtual

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

Mismo acceso
Mismo retorno
Mismo nombre
Mismos parámetros

Microsoft C#.net

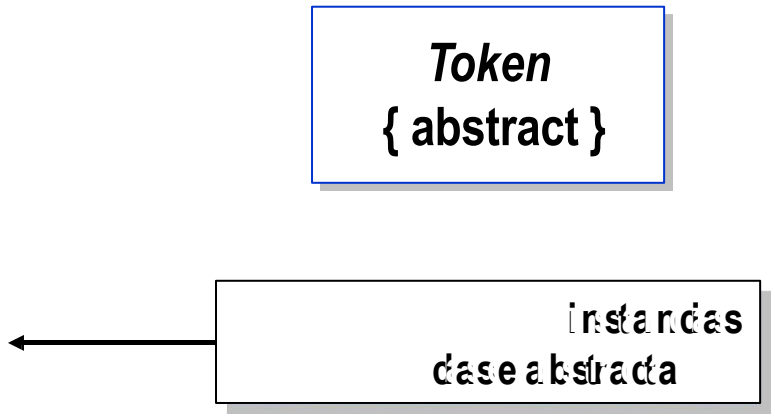
Uso de clases abstractas

- 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

Declaración de clases abstractas

- Se usa la palabra reservada abstract

```
abstract class Token
{
    ...
}
class Test
{
    static void Main( )
    {
        new Token( );3
    }
}
```



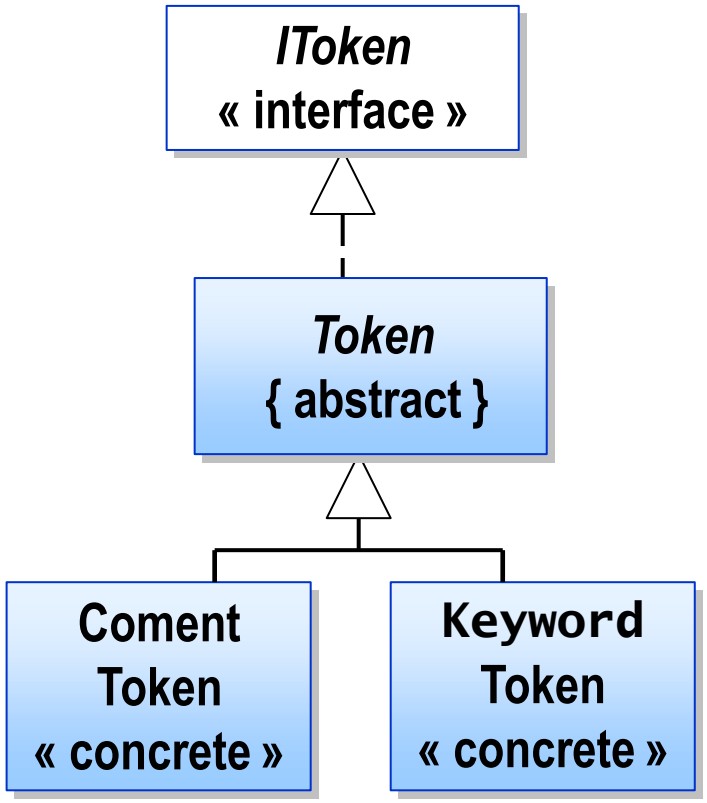
Uso de clases abstractas en una jerarquía de clases

- **Ejemplo 1**

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

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

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



C#.net

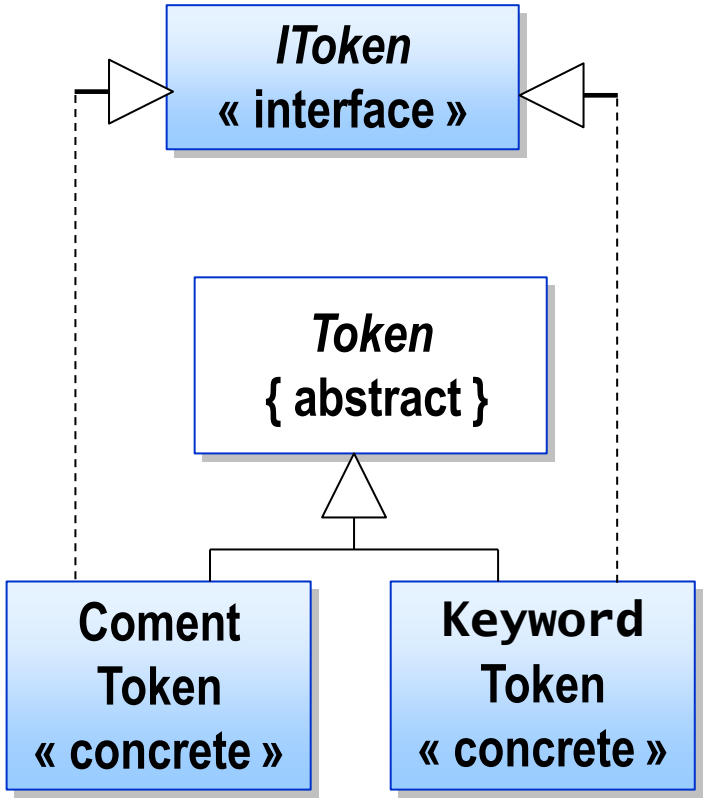
Uso de clases abstractas en una jerarquía de clases *(cont.)*

- **Ejemplo 2**

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

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

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



Comparación de clases abstractas e interfaces

- 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

Implementación de métodos abstractos

- 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( ) { ... }
}
```

- Sólo clases abstractas pueden declarar métodos abstractos
- Los métodos abstractos no pueden tener cuerpo

Microsoft C#.net

Uso de métodos abstractos

- 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

Propiedades

- Las propiedades son miembros que ofrecen un mecanismo flexible para leer, escribir o calcular los valores de campos privados.
- Se pueden utilizar las propiedades como si fuesen miembros de datos públicos, aunque en realidad son métodos especiales denominados descriptores de acceso.
- De este modo, se puede tener acceso a los datos con facilidad, a la vez que proporciona la seguridad y flexibilidad de los métodos.

Propiedades

- En este ejemplo, la clase TimePeriod almacena un período de tiempo.
- Internamente, la clase almacena el tiempo en segundos, pero se proporciona una propiedad denominada Hours que permite que un cliente especifique el tiempo en horas.
- Los descriptores de acceso de la propiedad Hours realizan la conversión entre horas y segundos.

```
class TimePeriod
{
   private double seconds;
   public double Hours
   {
      get { return seconds / 3600; }
      set { seconds = value * 3600; }
   }
}
```

Microsoft C#.net

Propiedades

```
class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();
        t.Hours = 24;
        System.Console.WriteLine("Time in hours: " +
t.Hours);
    }
}
```

- En este ejemplo, se puede ver el uso de la propiedad Hours, en modo escritura y lectura.

Propiedades

- Las propiedades proporcionan la comodidad de utilizar miembros de datos públicos sin los riesgos que implica el acceso no protegido y sin control ni comprobación a los datos de un objeto.
- Esto se logra a través de los descriptores de acceso: métodos especiales que asignan y recuperan los valores del miembro de datos subyacente.
- El descriptor de acceso **set** permite asignar los miembros de datos.
- El descriptor de acceso **get** recupera los valores de los miembros de datos.

Propiedades

```
class Person {
    private string m_name = "N/A";
    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return m_name;
        }
        set
        {
            m_name = value;
        }
    }
```

Delegados

- Un delegado es un tipo que hace referencia a un método.
- Cuando se asigna un método a un delegado, éste se comporta exactamente como el método.
- El método delegado se puede utilizar como cualquier otro método, con parámetros y un valor devuelto. Ejemplo:

  ```
  public delegate int PerformCalculation(int x, int y);
  ```

- Cualquier método que coincide con la firma del delegado, que está compuesta por los parámetros y el tipo de valor devuelto, puede asignarse al delegado.
- Esto permite el cambio mediante programación de las llamadas a métodos y la incorporación de nuevo código en las clases existentes.
- Si conoce la firma del delegado, puede asignar su propio método delegado.

Delegados

```
class Class1{
delegate double operacion(double param1, double param2);
static double Multiplica(double param1, double param2) {
return param1 * param2;
}
static double Divide(double param1, double param2) {
return param1 / param2;
    }

static void Main(string[] args) {
operacion op;
string input;
Console.Write("Escribe un número:");
input = Console.ReadLine();
double param1 = Convert.ToDouble(input);
```

Delegados

```
Console.Write("Escribe otro número:");
input = Console.ReadLine();
double param2 = Convert.ToDouble(input);
Console.WriteLine("(M)ultiplica o (D)ivide:");
input = Console.ReadLine();
if (input == "M")
op = new operacion(Multiplica);
else
op = new operacion(Divide);
Console.WriteLine("Result: {0}", op(param1, param2));
    }
   }
}
```

Microsoft C#.net

Delegados

- Los delegados tienen visibilidad.
- Los delegados se crean con el operador new.
- El constructor del delegado espera el nombre del método.
- Comprobación de tipos en tiempo de ejecución.
- Los delegados pueden guardar más de un método.
- La invocación de un delegado provoca la invocación de todos los métodos que almacena.

Delegados

```
public delegate void D();
class Class1 {
    void h1() { Console.WriteLine("Hola 1"); }
    void h2() { Console.WriteLine("Hola 2"); }
    void h3() { Console.WriteLine("Hola 3"); }
    static void Main(string[] args)
    {
        Class1 obj = new Class1();
        D delegado = new D(obj.h1);
        delegado += new D(obj.h2);
        delegado += new D(obj.h3);
        delegado();
        Console.ReadLine();
    }
}
```

Microsoft C#.net

Delegados

¿Qué hace el siguiente código?

```
class Class1 {
public delegate void M();
M mDel;
void m1() {
    Console.WriteLine("Llamada por primera vez.");
    this.mDel -= new M(this.m1);
    this.mDel += new M(this.m2);
    this.m2();
}
void m2() {
        Console.WriteLine("Realizando operación.");
   }
```

Microsoft C#.net

Delegados

```
void run() {
      this.mDel = new M(m1);
      mDel();
      mDel();
      mDel();
      Console.ReadLine();
}
static void Main(string[] args) {
    new Class1().run();
}
}
```

Bibliografía

- **C#. Curso de Programación**.
  - Autor: Fco. Javier Ceballos Sierra.
  - Editorial:
    - **RA-MA** en España.
    - **Alfaomega Grupo Editor** en América.
- **Enciclopedia de Microsoft Visual C#.**
  - Autor: Fco. Javier Ceballos Sierra.
  - Editorial:
    - **RA-MA** en España.
    - **Alfaomega Grupo Editor** en América.

Microsoft .net