Unidad

3

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 5

Clases: Conceptos Avanzados

En este Capítulo

- Tipos nulificables
- La palabra reservada static o Shared
- Acciones sobre la cadena de herencia
- Métodos recursivos
- Delegados
- Definición y uso de eventos
- Definiciones parciales
- Características especiales de las propiedades
- Indexadores y propiedades por defecto
- Sobrecarga de operadores
- La Clase Object
- Clases Abstractas
- Polimorfismo
- Interfaces
- Clases Anidadas

Tipos nulificables

Cuando se crea una variable por referencia, no tiene inicialmente asignada la dirección de ningún objeto. No se puede utilizar una variable por referencia hasta que se le haya asignado un valor, pero puede suceder que en ese punto del código en el que se declara la variable, no sepa con cuál valor inicializarla. En este caso, se puede establecer la variable por referencia en `null` o `Nothing` (C# o VB) para indicar que no se ha inicializado. Análogamente, cuando se quiere desechar un objeto determinado y convertirlo en seleccionable para el garbage collector o no se quiera utilizar más en el código, puede asignarse a la referencia un valor nulo.

El valor nulo es útil porque se lo puede comprobar explícitamente más adelante en el código y, si una variable de referencia es nula, se la puede inicializar utilizando el operador `new` o `New` (C# o VB) como muestra el siguiente ejemplo de código.

Ejemplo

C#

```
MiClase m = new MiClase();
m = null;
```

VB

```
Dim m As New MiClase()
m = Nothing
```

Donde

C#

```
namespace nulificables
{
    class MiClase
    {
        public void UnMetodo(int a)
        {
        }

        public void OtroMetodo(int? a)
        {
            if (a.HasValue)
                Console.WriteLine("El valor es: " + a.Value);
        }
    }
}
```

VB

```
Public Class MiClase
    Public Sub UnMetodo(a As Integer)

    End Sub

    Public Sub OtroMetodo(a As Integer?)
```

```
        If (a.HasValue) Then
            Console.WriteLine("El valor es: " + a.Value.ToString)
        End If
        Console.ReadKey()
    End Sub
End Class
```

El valor nulo es en sí mismo una referencia en C#. No existe un valor correspondiente en los tipos por valor.

Por otro lado en VB, asignar a un tipo por valor `Nothing` causa una asignación del valor por defecto del tipo. Esto puede derivar en problemas de interpretación. Los valores asignados con `Nothing` tienen un significado diferente si es un tipo por valor o por referencia. En los tipos por valor causa una asignación del valor por defecto del tipo, mientras que en los tipos por referencia causa una asignación de referencia nula. ***En VB es válido asignar un*** `Nothing` ***a un tipo por valor***.

Esto puede causar un problema en el código. Por ejemplo, puede que no sea fácil determinar si un tipo de valor se haya inicializado (recuerde que si se intenta pasar un tipo por valor sin inicializar en un método, el código no compilará en C#, pero si en VB).

Ejemplo

C#

```
int a;
m = new MiClase();
m.UnMetodo(a);
```

VB

```
Dim a As Integer
m = New MiClase()
m.UnMetodo(a)
```

Mostrará el siguiente error en C#, pero no en VB:

> Use of unassigned local variable 'a'

Debido a que el valor nulo se trata como a una referencia en C#, la instrucción que se muestra en el siguiente ejemplo de código es ilegal en C #.

Ejemplo

C#

```
a = null;
```

El error obtenido es:

> Cannot convert null to 'int' because it is a non-nullable value type

Sin embargo, C# y VB definen un modificador que se puede utilizar para declarar que una variable es un tipo por valor que acepta valores `null` o `Nothing` (C# o VB). Un tipo por valor que acepta valores `null` o `Nothing` (C# o VB) se comporta de manera similar al tipo por valor original, pero con el adicional que se puede asignar el valor nulo a la misma. Se utiliza un signo de interrogación (?) para indicar que un tipo por valor se puede nulificar. Más tarde, en la aplicación, se puede comprobar si una variable nulificable contiene `null` o `Nothing` (C# o VB). Para ello, de la misma manera como con un tipo por referencia, se puede verificar si su contenido es `null` o `Nothing` (C# o VB), como muestra el ejemplo de código siguiente.

Ejemplo

C#

```
int? b = null;

if (b == null)
{
        b = 5;
        m.UnMetodo(b);
}
```

VB

```
Dim b As Integer? = Nothing

Console.WriteLine("El valor de b = Nothing es " + b.ToString)

If b Is Nothing Then
        b = 5
        m.UnMetodo(b)
End If
```

Sin embargo este código también genera un error en C#, ***pero no en VB***. La razón es que se intenta asignar un tipo nulificable a uno que no lo es, lo cual genera el siguiente error en tiempo de compilación:

> Argument 1: cannot convert from 'int?' to 'int'

Por otro lado si se genera un método con un tipo nulificable como parámetro, el problema se soluciona en C# y se comporta análogamente al caso anterior en VB.

Ejemplo

C#

```
if (b == null)
{
        b = 5;
        m.OtroMetodo(b);
}
```

VB

```
b = Nothing
If b Is Nothing Then
    b = 5
    m.OtroMetodo(b)
End If
```

Como la variable y el parámetro pueden contener valores nulos, la asignación debida al argumento estará bien definida.

Propiedades de los tipos que aceptan valores nulos

Los tipos nulificables exponen un par de propiedades que se pueden utilizar para determinar si una variable declarada para aceptar valores nulos tiene un valor diferente a éste o no:

- **HasValue**. Esta es una propiedad booleana que indica si un tipo que acepta valores nulos contiene un valor diferente a éste o no. Si esta propiedad es `true` o `True` (C# o VB), la variable nulificable tiene un valor, si es falsa, la almacena un `null` o `Nothing` (C# o VB).
- **Value**. Este es el valor que almacena una variable nulificable. Sólo se debe intentar leer este valor si la propiedad HasValue es `true` o `True` (C# o VB), de lo contrario el código producirá una excepción (error). Esta propiedad es de sólo lectura.

En el ejemplo de código siguiente se muestra cómo utilizar estas propiedades con una variable.

Ejemplo

C#

```
public void OtroMetodo(int? a)
{
        if (a.HasValue)
                Console.WriteLine("El valor es: " + a.Value);
}
```

VB

```
Public Sub OtroMetodo(a As Integer?)
        If (a.HasValue) Then
                Console.WriteLine("El valor es: " + a.Value.ToString)
        End If
End Sub
```


La palabra reservada `static` o `Shared`

Las clases poseen los modificadores de visibilidad ya enunciados para la declaración de sus elementos (privado, público, protegido y visibilidad de ensamblado). Sin embargo existen modificadores de visibilidad adicionales que afectan radicalmente el uso de un atributo, método o clase anidada. Uno de ellos es `static` o `Shared` (C# o VB).

Cuando un atributo o método se declara `static` o `Shared` (C# o VB), indica que la visibilidad esta dentro de la clase, no de una instancia de esta, por eso se los denomina métodos o atributos “de la clase” en lugar del objeto. Esta definición, si bien no es muy afortunada, tiene su origen en el

hecho que no se necesita un objeto del tipo de la clase para acceder al elemento declarado como tal, pero se debe indicar como accederla a través de un nombre totalmente calificado por notación de punto que incluye el nombre de la clase en la que esta definido.

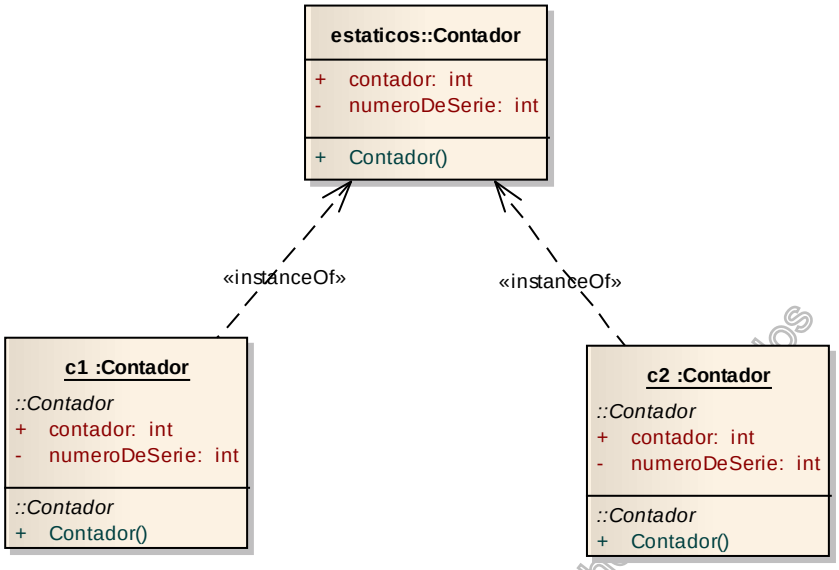
Como no se necesita un objeto del tipo de la clase, indica claramente que estos elementos no pertenecerán al objeto que se crea a partir de la clase que los contiene y por lo tanto no podrán acceder tampoco a los miembros de un objeto simplemente "porque no los ve". La razón de fondo para esto es que al indicar el modificador de visibilidad `static` o `Shared` (C# o VB) se especifica al lenguaje que no asocie una referencia de tipo `this` o `Me` (C# o VB) al elemento, y, como se mencionó anteriormente, este es el medio por el cual se accede a los miembros de un objeto en tiempo de ejecución.

Atributos `static` o `Shared`

Un atributo `static` o `Shared` (C# o VB) se almacena en la memoria estática, por lo tanto, no se pueden crear atributos de este tipo cada vez que se crea un objeto.

Entonces, ¿cómo soluciona esta limitación el lenguaje? La respuesta en realidad es simple: se crea una sola vez el atributo en la memoria estática y dicho lugar de almacenamiento se compartirá por todos los objetos del mismo tipo, o, en otras palabras, todos los objetos del tipo de la clase en donde se declaró el atributo pueden leer y escribir en él. Más aún, si el atributo tiene un modificador de visibilidad que determine que éste sea visible (público, protegido o de ensamblado según sea el caso), con resolver el nombre totalmente calificado (nombreDeClase.nombreDeAtributo) podrá ser accedido por cualquier clase.

A continuación, y como ejemplo, se muestra el diagrama de una clase en UML junto con dos objetos, c1 y c2, que son instancias del tipo Contador. Notar que el estereotipo del vínculo en el diagrama (instanceOf) indica que son dos objetos del tipo Contador. Además, c1 y c2 son diagramas de objetos mientras que Contador es de clase. Notar también que, en esta última, aparece el atributo contador subrayado y en UML esto indica que el atributo es estático (también llamado *compartido*)



La forma de acceso a un atributo estático desde otra clase se muestra en el siguiente código.

Ejemplo

C#

```
namespace contador
{
    public class Contador
    {
        public static int contador = 0;
        private int numeroDeSerie;

        public Contador()
        {
            contador++;
            numeroDeSerie = contador;
        }
    }
}
```

```
namespace contador
{
    class Program
    {
        static void Main(string[] args)
        {
            Contador c1 = new Contador();
```

```
            Console.WriteLine("El valor de contador es {0}" , Contador.contador);
            Contador c2 = new Contador();
            Console.WriteLine("El valor de contador es {0}", Contador.contador);
            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class Contador
    Public Shared contador As Integer = 0
    Private numeroDeSerie As Integer

    Public Sub New()
        contador += 1
        numeroDeSerie = contador
    End Sub
End Class



Module Module1
    Sub Main()
        Dim c1 As New Contador
        Console.WriteLine("El valor de contador es {0}", Contador.contador)
        Dim c2 As New Contador()
        Console.WriteLine("El valor de contador es {0}", Contador.contador)
        Console.ReadKey()
    End Sub
End Module
```


Notar que al estar el atributo contador en el constructor y al ser estático, cada vez que se cree un objeto ocasionará que se incremente el valor almacenado en esa área de memoria. Como todos los objetos de tipo Contador comparten esta área de memoria, cada vez que se cree un objeto se incrementará sobre valor al ejecutar el constructor, lo cual deviene “en una especie” de contador de objetos del mismo tipo. La salida producida por el programa es la siguiente:

El valor de contador es 1
El valor de contador es 2

Métodos `static` o `Shared`

Se puede invocar un método `static` o `Shared` (C# o VB) sin una instancia de la clase a la que pertenezca. La razón de esto es que el método no tiene un `this` o `Me` (C# o VB) asociado, lo cual implica que no pertenece a la instancia de ningún objeto del tipo de la clase. Sin embargo, esta contenido por la clase y esto implica que se deberá resolver la visibilidad por medio de un nombre totalmente calificado con notación de punto para accederlo.

Suponiendo que se declara una variable de instancia como privada, no podrá ser accedida desde fuera de la clase con notación de punto debido a la declaración de su modificador. En casos como

estos, la única forma de acceder a variables con este tipo de declaraciones es mediante un método estático.

Modificando levemente el ejemplo anterior, se puede apreciar este hecho.

Ejemplo

C#

```
namespace estaticos
{
    public class Contador
    {
        private static int contador = 0;
        private int numeroDeSerie;

        public Contador()
        {
            contador++;
            numeroDeSerie = contador;
        }

        public static int GetContador()
        {
            return contador;
        }

        public static void IncrementarContador()
        {
            contador++;
        }
    }
}

namespace estaticos
{
    public class OtraClase
    {
        public void incrementarNumero()
        {
            Contador.IncrementarContador();
        }
    }
}
```

VB

```
Public Class Contador
    Public Shared contador As Integer = 0
    Private numeroDeSerie As Integer

    Public Sub New()
        contador += 1
        numeroDeSerie = contador
    End Sub
```

```
    Public Shared Function GetContrador() As Integer
        Return contador
    End Function

    Public Shared Sub IncrementarContador()
        contador += 1
    End Sub
End Class


Public Class OtraClase
    Public Sub IncrementarNumero()
        Contador.IncrementarContador()
    End Sub
End Class
```

Por lo tanto, si se quiere acceder el método, se deberá resolver la visibilidad de acceso al mismo con un nombre totalmente calificado que incluya al de la clase. El siguiente ejemplo muestra esta situación.

Ejemplo

C#

```
namespace estaticos
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("El valor de contador es " + Contador.GetContador());
            Contador c = new Contador();
            Console.WriteLine("El valor de contador es " + Contador.GetContador());
            OtraClase o = new OtraClase();
            o.incrementarNumero();
            Console.WriteLine("El valor de contador es " + Contador.GetContador());
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
    Sub Main()
        Console.WriteLine("El valor de contador es " + _
              Contador.GetContrador().ToString)
        Dim c As New Contador()
        Console.WriteLine("El valor de contador es " + _
              Contador.GetContrador().ToString)
        Dim o As New OtraClase()
        o.IncrementarNumero()
```

```
        Console.WriteLine("El valor de contador es " + _
              Contador.GetContrador().ToString)
        Console.ReadKey()
    End Sub
End Module
```

La salida del programa es:

El valor de contador es 0
El valor de contador es 1
El valor de contador es 2

El patrón de diseño Singleton (instancia única)

Este patrón de diseño tiene la característica que sólo existirá un objeto de su tipo lo largo de toda la ejecución del código en el que se encuentre definido. Para lograr esta implementación en tecnología .Net, se puede aprovechar la característica de los elementos estáticos de una clase.

Una declaración `static` o `Shared` (C# o VB) significa que cuando el cargador de clases del CLR detecte el uso de un atributo de este tipo, el mismo será el primero en estar disponible en memoria, **inclusive antes que se cree cualquier otro objeto**. Por otra parte, los métodos `static` o `Shared` (C# o VB) deben poder invocarse antes que cualquier otro método, por lo tanto el CLR deja disponible la posibilidad de invocarlos al comienzo de cada programa que los incluya.

Ejemplo

C#

```
namespace singleton
{
    public class InstanciaUnica
    {
        private static InstanciaUnica _instanciaUnica = new InstanciaUnica();

        private InstanciaUnica()
        {
            Console.WriteLine("Creando instancia única");
        }

        public static InstanciaUnica GetInstanciaUnica()
        {
            return _instanciaUnica;
        }

        public void OtroMetodo()
        {
            Console.WriteLine("Estoy en OtroMetodo");
        }

        public void OtroMetodo2()
        {
            Console.WriteLine("Estoy en OtroMetodo2");
```

```
        }
    }
}
```

VB

```
Public Class InstanciaUnica
    Private Shared _instanciaUnica As New InstanciaUnica
    Private Sub New()
        Console.WriteLine("Creando instancia única")
    End Sub

    Public Shared Function GetInstanciaUnica() As InstanciaUnica
        Return _instanciaUnica
    End Function

    Public Sub OtroMetodo()
        Console.WriteLine("Estoy en OtroMetodo")
    End Sub

    Public Sub OtroMetodo2()
        Console.WriteLine("Estoy en OtroMetodo2")
    End Sub
End Class
```

El secreto radica en los siguientes pasos:

1. Existe un atributo `static` o `Shared` (C# o VB) en la clase InstanciaUnica y el CLR lo pone en memoria antes que nada.
2. Como el atributo tiene una asignación, trata de resolverla para así terminar con la declaración.
3. Cuando trata de inicializar se encuentra con un operador `new` o `New` (C# o VB) y lo resuelve.
4. El operador actúa sobre la misma clase InstanciaUnica, y, a pesar que el constructor de la clase InstanciaUnica es privado, lo ve por estar definido en la clase
5. Como el hecho es tratar de dejar disponible un objeto del tipo InstanciaUnica, se debe poseer una forma de devolver la referencia en un método que pertenezca a la clase para que sea accesible en cualquier situación que pudiera existir, por lo tanto, no puede depender de la creación del objeto. Como este método es la única forma de obtener una referencia, siempre se accede por él y como retorna un atributo estático que no pertenece a un determinado objeto, se puede invocar al método “siempre” para obtener la referencia

Un ejemplo de uso se presenta a continuación.

Ejemplo

C#

```
namespace singleton
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            InstanciaUnica si = InstanciaUnica.GetS();
            si.OtroMetodo();
            si.OtroMetodo2();
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim si As InstanciaUnica = InstanciaUnica.GetInstanciaUnica()
        si.OtroMetodo()
        si.OtroMetodo2()
        Console.ReadKey()
    End Sub
End Module
```

Se debe notar que una vez obtenida la referencia, se puede acceder a cualquier otro método definido dentro de la clase porque dicho objeto ya existe. La salida producida por el programa es al siguiente:

Creando instancia única
Estoy en OtroMetodo
Estoy en OtroMetodo2

Acciones sobre la cadena de herencia

Se pueden definir acciones que cambian la manera en que se puede manejar el comportamiento por defecto en una cadena de herencia. Dos acciones que se pueden alterar son las de finalizar la capacidad de heredar y la de rescribir un método. A esto se lo llama "sellar". Por lo tanto, los elementos de C# y VB que se ven afectados por dichas acciones son:

- Un método
- Una clase

Métodos

Cuando el diseño de una clase demanda que un método no sea sobrescrito por las subclases, se debe omitir la declaración `virtual` u `Overridable` (C# o VB) en el mismo. Al no tener este modificador de rescritura, cualquier método que lo intente rescribir generará un error.

Ejemplo

C#

```
namespace sellados
{
```

```
    public class Super
    {
        internal int unNumero;
        public void met() { }
        public virtual void met2(){}
    }
}
```

VB

```
Public Class Super
    Friend unNumero As Integer = 0
    Public Sub met()

    End Sub
    Public Overridable Sub met2()

    End Sub
End Class
```

La clase Super declara un método llamado met() que no incluye el modificador `virtual` u `Overridable` (C# o VB). Si se intenta rescribir el método generará un error.

Ejemplo

C#

```
namespace sellados
{
    public class Sub : Super
    {
        internal new int unNumero=1;

        public Sub()
        {
            int ej = base.unNumero;
        }

        public sealed override void met2()
        {
            base.met2();
        }

        public override void met()
        { // ERROR

        }
    }
}
```

VB

```
Public Class SubC
    Inherits Super

    Friend Shadows unNumero As Integer = 1
```

```
    Public Sub New()
        Dim ej As Integer = MyBase.unNumero
    End Sub

    Public NotOverridable Overrides Sub met2()
        MyBase.met2()
    End Sub

    Public Overrides Sub met()
        'ERROR
    End Sub
End Class
```

El error en tiempo de compilación es el siguiente:

C#

'sellados.Sub.met()': cannot override inherited member 'sellados.Super.met()' because it is not marked virtual, abstract, or override

VB

'Public Overrides Sub met()' cannot override 'Public Sub met()' because it is not declared 'Overridable'.

Otra situación de diseño que se puede presentar es que un determinado método no deba ser rescrito a partir de una determinada clase en una cadena de herencias. En este caso, la superclase declaró el método como `virtual` u `Overridable` (C# o VB) pero se quiere finalizar a partir de una determinada clase la capacidad de rescritura. En el ejemplo anterior, la clase Super declara el método met2() como `virtual` u `Overridable` (C# o VB)y la subclase Sub lo rescribe correctamente, pero agrega un modificador: `sealed` u `NotOverridable` (C# o VB).

Ejemplo

C#

```
namespace sellados
{
    class SubSealed : Super
    {
        public sealed override void met2() { }
    }
}
```

VB

```
Public Class SubNotOverridable
    Inherits Super
    Public NotOverridable Overrides Sub met2()

    End Sub
End Class
```

Una subclase no puede sobrescribir métodos que hayan sido declarados de esta manera en la superclase (por definición, los métodos `sealed` u `NotOverridable` -C# o VB- no pueden ser sobrescritos).

Ejemplo

C#

```
namespace sellados
{
    class SubSubSealed : SubSealed
    {
        public override void met2() { }
    }
}
```

VB

```
Public Class SubSubNotOverridable
    Inherits SubNotOverridable
    Public Overrides Sub met2()

    End Sub

End Class
```

Si se intenta sobrescribir un método de este tipo, el compilador mostrará un mensaje de error como el siguiente y no compilará el programa.

C#

'sellados.SubSubSealed.met2()': cannot override inherited member 'sellados.SubSealed.met2()' because it is sealed

VB

'Public Overrides Sub met2()' cannot override 'Public Overrides NotOverridable Sub met2()' because it is declared 'NotOverridable'.

Temas a tener en cuenta al querer utilizar métodos con la declaración `sealed` u `NotOverridable` (C# o VB):

- Al definir nuevos métodos o propiedades en una clase, se puede evitar que las clases derivadas lo sobrescriban (rescritura) al no declararlos como `virtual` u `Overridable` (C# o VB).
- Cuando se aplica a un método o propiedad, el modificador de `sealed` u `NotOverridable` (C# o VB) debe ser siempre utilizado con `override` u `Overrides` (C# o VB), sino se generará un error.
- Debido a que las estructuras se son `sealed` u `NotOverridable` (C# o VB) implícitamente, no se pueden heredar.

- Una subclase tampoco pude sobrescribir métodos que se hayan declarado como `static` o `Shared` (C# o VB) en la superclase. En otras palabras, una subclase no puede sobrescribir un método de clase.
- Los métodos `static` o `Shared` (C# o VB) y `private` o `Private` (C# o VB) son sellados automáticamente

Se podría hacer que un método fuera `sealed` si el método tiene una implementación que no debe ser cambiada y que es crítica para el estado consistente del objeto a instanciar.

Clases

Se puede declarar que una clase sea `sealed` u `NotInheritable` (C# o VB), lo que implica que la clase no puede tener subclases.
Se realiza por motivos de diseño o seguridad y tiene un formato como el siguiente:

Ejemplo

C#

```
namespace sellados
{
    public sealed class SuperSealed
    {
        public void met4()
        {
        }
    }
}
```

VB

```
Public NotInheritable Class SuperNotInheritable
    Public Sub met4()

    End Sub
End Class
```

Cualquier intento posterior de crear una subclase de SuperSealed o SuperNotInheritable (C# o VB) resultará en un error del compilador.

Ejemplo

C#

```
namespace sellados
{
    class SubSuperSealed : SuperSealed
    {
    }
}
```

VB

```
Public Class SubSuperNotInheritable
    Inherits SuperNotInheritable
```

```
End Class
```

El error que se obtendrá es como el siguiente:

C#

'sellados.SubSuperSealed': cannot derive from sealed type 'sellados.SuperSealed'

VB

'SubSuperNotInheritable' cannot inherit from class 'SuperNotInheritable' because 'SuperNotInheritable' is declared 'NotInheritable'.

Seguridad

Un mecanismo que los hackers utilizan para atacar sistemas es crear subclases de una clase y luego sustituirla por el original. Las subclases se parecen y sienten como la clase original pero hacen cosas bastante diferentes, probablemente causando daños u obteniendo información privada. Para prevenir esta clase de modificación del código, se puede declarar que la clase sea `sealed` u `NotInheritable` (C# o VB) y así prevenir que se cree cualquier subclase de ella.

La clase String del espacio de nombres System es una clase `sealed` u `NotInheritable` (C# o VB) sólo por esta razón. Esta es tan vital para la operación del CLR que debe garantizar siempre que un método u objeto que utilice un String, obtenga un objeto System.String y no algún otro tipo derivado de String. Esto asegura que ningún String tendrá propiedades extrañas, inconsistentes o indeseables.

Diseño

Otra razón por la que se podría querer declarar una clase `sealed` u `NotInheritable` (C# o VB) son razones de diseño orientado a objetos. Se podría pensar que una clase es "perfecta" o que, conceptualmente hablando, la clase no debería tener subclases.

Métodos recursivos

Existen técnicas de programación de algoritmos que necesitan que un método se llame a sí mismo para realizar una acción que siempre es la misma un número determinado de veces. Como cada método antes de ejecutarse pide su propio espacio en el stack, se debe ser cauto de tener un mecanismo adecuado que determine el fin de las “autollamadas” antes que se agote el mismo (lo que se conoce como desbordamiento de pila o stack).

En este punto no es un objetivo mostrar técnicas de algoritmos, pero si mostrar como queda esta solución en un ambiente de objetos. Para ello se diseñó un ejemplo que presenta dos problemas típicos de llamados recursivos de métodos: el cálculo del factorial de un número y el resultado de la suma de la serie de Fibonacci.

Ejemplo

C#

```
namespace recursivos
{
    class MetodosRecursivos
    {
        public long CalculaFactorial(int i)
        {
            return ((i <= 1) ? 1 : (i * CalculaFactorial(i - 1)));
        }

        public int Fib(int n)
        {
            if (n < 2)
                return n;
            else
                return Fib(n - 1) + Fib(n - 2);
        }
    }
}
```

```
namespace recursivos
{
    class Program
    {
        static void Main(string[] args)
        {
            MetodosRecursivos m = new MetodosRecursivos();

            Console.WriteLine("El factorial de 10 es: {0}", m.CalculaFactorial(10));
            Console.WriteLine("La serie de Fibonacci de 10 es: {0}", m.Fib(10));
            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class MetodosRecursivos

    Public Function CalculaFactorial(i As Integer)
        If i <= 1 Then
            Return 1
        Else
            Return i * CalculaFactorial(i - 1)
        End If
    End Function

    Public Function fib(ByVal n) As Integer
        If n < 2 Then
            Return n
        Else
```

```
            Return fib(n - 1) + fib(n - 2)
        End If
    End Function
End Class


Module Module1
    Sub Main()
        Dim m As New MetodosRecursivos()

        Console.WriteLine("El factorial de 10 es: {0}", m.CalculaFactorial(10))
        Console.WriteLine("La serie de Fibonacci de 10 es: {0}", m.fib(10))
        Console.ReadKey()
    End Sub
End Module
```

Delegados

Un delegado es una declaración que debe tratarse como la de un tipo (al igual que las clases y estructuras). La declaración de un delegado especifica una firma de método (también denominado prototipo) que tiene definido un conjunto de argumentos y un tipo retornado. Cuando se define un delegado, el compilador genera una clase de tipo seguro (esto es, el tipo se mantiene más allá de la plataforma en la cual se ejecute el programa) que encapsula la firma del método y permite al código de cliente invocar métodos que tienen la firma especificada en él.

Se lo considera un tipo porque tiene las mismas características que éstos, es decir, puede declararse una variable con él, pasar esta como argumento de una función o ser retornado por una de ellas.

Definición de delegados

Para definir un tipo de delegado, utilizar la palabra clave `delegate` o `Delegate` (C # o VB). La definición puede encontrarse en el ámbito de un espacio de nombres o en el de una clase ya que define un tipo en sí mismo. Esto quiere decir, que es un tipo definido en el espacio de nombres o está agregado o compuesto en una clase.

Los ejemplos de código siguientes se muestran cómo definir un tipo delegado. Se muestran los casos para los cuales los delegados retornan valor o no. Tener en cuenta que en VB los métodos que no retornan valores se definen como `Sub`.

Ejemplo

C#

```
public delegate int DelegadoFibonacci(int valor);
public delegate void DelegadoContador();
```

VB

```
Public Delegate Function DelegadoFibonacci(valor As Integer) As Integer
```

```
Public Delegate Sub DelegadoContador()
```

Creando una instancia de un delegado

Después de declarar un tipo de delegado, se puede crear un objeto delegado y asociarlo en el mismo u otro objeto (al declararlo en una clase o espacio de nombres) con un determinado método. Tener en cuenta que los delegados se ven afectados por las declaraciones de modificadores de visibilidad, por lo tanto el hecho de declararlos en otros objetos o clases dependerá de su nivel de acceso definido por dicho modificador.

Creación de un objeto delegado

Para crear un objeto delegado, utilizar el operador `new` o `New` (C # o VB) y especificar el nombre de un método como parámetro. Si se está utilizando VB, se debe agregar el nombre de la firma del método con el operador `AddressOf`.

Los ejemplos de código siguientes muestran cómo crear un objeto delegado mediante el uso del operador `new` o `New` (C # o VB) para encapsular el método `MetodoConContadorYMensaje`.

Ejemplo

C#

```
_metodoConContador = new DelegadoContador(MetodoConContadorYMensaje);
```

VB

```
_metodoConContador = New DelegadoContador(AddressOf MetodoConContadorYMensaje)
```

A partir de la versión 2.0 del Framework de .NET se permite utilizar una notación abreviada para crear objetos delegados. Se puede declarar una variable de delegado y asignarle un nombre de método directamente. De esta manera no se tiene que crear el nuevo objeto delegado explícitamente. El compilador inserta código para crear el objeto implícitamente.

Los ejemplos de código siguientes muestran cómo crear un objeto delegado implícitamente para encapsular el método `MetodoConContadorYMensaje`.

Ejemplo

C#

```
_metodoConContador = MetodoConContadorYMensaje;
```

VB

```
_metodoConContador = AddressOf MetodoConContadorYMensaje
```

Invocar un método utilizando una instancia de delegado

Después de crear un objeto delegado que hace referencia a un método en particular en un objeto o una clase, puede utilizar el objeto delegado para invocar el método de forma sincrónica o asincrónica.

Nota: el uso de delegados asíncronos se explicará posteriormente en el capítulo de threads

La invocación de un delegado se puede realizar por un método miembro del objeto llamado Invoke o utilizando directamente como cualquier llamado a función mediante el nombre que se haya asignado. En el siguiente ejemplo, se declaró un delegado como propiedad de una clase y se lo recupera en una variable referencia declarada mediante el mismo (recordar que al ser un tipo, se pueden definir variables del tipo del delegado) y se lo utiliza como si fuera un simple llamado a función. La clase que define al delegado asigna la firma de un método privado al mismo llamado MetodoConContadorYMensaje. La dirección en dónde se encuentra el método coincide con la firma declarada para el tipo DelegadoContador, o sea, es un método que no recibe parámetros ni retorna valores (los comentarios en el código indican el orden de construcción del programa).

Ejemplo

C#

```
namespace delegados
{
    public delegate int DelegadoFibonacci(int valor);
    public delegate void DelegadoContador();

    public class Delegado
    {
        //Paso 1: Definir el tipo de firma de métodos a publicar
        public delegate string FirmaMetodoAInvocar(string texto);

        //Paso 2: Definir la variable para guardar las direcciones
        //de las funciones a invocar
        private FirmaMetodoAInvocar metodoAInvocar;
        private DelegadoContador _metodoConContador;
        private int contadorDeLlamados = 0;

        public Delegado()
        {
            _metodoConContador = MetodoConContadorYMensaje;
        }

        public void AgregaFirmaAlDelegado(FirmaMetodoAInvocar metodoAInvocar)
        {
            this.metodoAInvocar += metodoAInvocar;
        }

        public void QuitaFirmaAlDelegado(FirmaMetodoAInvocar metodoAInvocar)
        {
            this.metodoAInvocar -= metodoAInvocar;
        }

        public void InvocaDelegados(string texto)
        {
            //Paso 6: Invocar las funciones
```

```
            Console.WriteLine(metodoAInvocar(texto) + Environment.NewLine);
        }

        public void CalculaSerieDeFibonacci(int n)
        {
            Fibonacci f = new Fibonacci();
            DelegadoFibonacci df = f.MetodoParaCalcular;
            Console.WriteLine("La serie de Fibonacci para {0} es: {1}", n, df(n));
        }

        private void MetodoConContadorYMensaje()
        {
            contadorDeLlamados++;
            Console.WriteLine("Este es el llamado Nro: " + contadorDeLlamados);
        }
        public DelegadoContador MetodoConContador
        {
            get
            {
                return _metodoConContador;
            }
        }
    }
}
```

VB

```
Public Delegate Function DelegadoFibonacci(valor As Integer) As Integer
Public Delegate Sub DelegadoContador()

Public Class Delegado
    'Paso 1: Definir el tipo de firma de métodos a publicar
    Public Delegate Function FirmaMetodoAInvocar(texto As String)

    'Paso 2: Definir la variable para guardar las direcciones
    'de las funciones a invocar
    Private metodoAInvocar As FirmaMetodoAInvocar
    Private _metodoConContador As DelegadoContador
    Private contadorDeLlamados As Integer = 0

    Public Sub New()
        _metodoConContador = AddressOf MetodoConContadorYMensaje
    End Sub

    Public Sub AgregaFirmaAlDelegado(metodoAInvocar As FirmaMetodoAInvocar)
        Me.metodoAInvocar = [Delegate].Combine(Me.metodoAInvocar, metodoAInvocar)
    End Sub

    Public Sub QuitaFirmaAlDelegado(metodoAInvocar As FirmaMetodoAInvocar)
        Me.metodoAInvocar = [Delegate].Remove(Me.metodoAInvocar, metodoAInvocar)
    End Sub

    Public Sub CalculaSerieDeFibonacci(n As Integer)
        Dim f As New Fibonacci()
        Dim df As DelegadoFibonacci = f.MetodoParaCalcular
        Console.WriteLine("La serie de Fibonacci para {0} es: {1}", n, df(n))
```

```
    End Sub

    Public Sub InvocaDelegados(texto As String)
        'Paso 6: Invocar las funciones
        Console.WriteLine(metodoAInvocar(texto) + Environment.NewLine)
    End Sub
    Private Sub MetodoConContadorYMensaje()
        contadorDeLlamados += 1
        Console.WriteLine("Este es el llamado Nro: " + contadorDeLlamados.ToString)
    End Sub
    Public ReadOnly Property MetodoConContador() As DelegadoContador
        Get
            Return _metodoConContador
        End Get
    End Property
End Class
```

Notar que la asignación de la firma del método privado se realiza en el constructor de la clase.

La propiedad se utiliza para obtener el delegado y así realizar la invocación al método en el objeto con el que se está tratando (los comentarios en el código indican el orden de construcción del programa). El delegado se almacena en la referencia "metodo"

Ejemplo

C#

```
        // Paso 3: Crear los objetos. Uno tiene el delegado,
        // el otro el método a asignar al delegado
        Delegado unObjetoQueTieneDelegados = new Delegado();
        UnaClase unObjetoConUnMetodo = new UnaClase();
        UnaClase unObjetoConUnMetodo2 = new UnaClase();
        OtraClase unObjetoConUnMetodo3 = new OtraClase();

        // Paso 4: Obtener el delegado a un método privado en la clase Delegado
        DelegadoContador metodo = unObjetoQueTieneDelegados.MetodoConContador;

        // Paso 5: Invocar al método privado en la clase Delegado como si fuera
        // público
        metodo();
        metodo();
        metodo();
        metodo();
```

VB

```
        ' Paso 3: Crear los objetos. Uno tiene el delegado,
        ' el otro el método a asignar al delegado
        Dim unObjetoQueTieneDelegados As New Delegado()
        Dim unObjetoConUnMetodo As New UnaClase()
        Dim unObjetoConUnMetodo2 As New UnaClase()
        Dim unObjetoConUnMetodo3 As New OtraClase()

        ' Paso 4: Obtener el delegado a un método privado en la clase Delegado
        Dim metodo As DelegadoContador = unObjetoQueTieneDelegados.MetodoConContador
```

```
' Paso 5: Invocar al método privado en la clase Delegado como si fuera
' público
metodo()
metodo()
metodo()
metodo()
```

Nota: el ejemplo muestra un posible uso INCORRECTO de delegados. La razón de ello es que rompe el encapsulamiento porque es una referencia directa al área de memoria donde reside el mismo sin tener en cuenta el modificador de visibilidad privado con el que se lo declaró. Por esta causa, **no se deben utilizar delegados de esta forma**.

Llamada a múltiples métodos con un delegado (multidifusión – multicast)

El Framework de .Net admite la combinación de varias llamadas a métodos dentro de un mismo delegado. La razón de ello es que muchas veces ante un determinado suceso, se quieren invocar varios métodos a la vez.

Esto se logra porque en realidad existe una clase que define a los tipos delegados llamada MulticastDelegate que es subclase de Delegate. Ambas clases definidas en el Framework de .Net no pueden ser heredadas en un programa común sino sólo en la creación de herramientas mediante un uso particular de las mismas. Sin embargo, su utilización es transparente para el programador y le permite agrupar varias firmas de métodos en un mismo delegado con el formato interno de una lista enlazada (razón por la cuál, la secuencia de invocación es en el mismo orden en el cual se agregaron las firmas de los métodos a invocar). Cuando se realiza la llamada mediante el delegado, todos los métodos cuyas firmas se agregaron al delegado de multidifusión serán invocados.

Para agregar firmas de métodos las técnicas varían según el lenguaje:

- En C# se utiliza el operador += para incorporar desde la primera firma a la última. Se admite su uso aunque se agregue una sola firma. También se puede utilizar el método Combine que se explica para VB, pero este el formato más difundido.
- En VB, se utiliza el método Combine que recibe dos argumentos, el primero con el delegado que se utiliza en primer lugar de la lista, el segundo con el último delegado que se agregará a la lista. Este método retorna el delegado de multidifusión a utilizar. El mismo método también utiliza un formato con un solo argumento para lograr el mismo objetivo.

El siguiente código es un extracto de la clase Delegado mostrada con anterioridad.

Ejemplo

C#

```
public void AgregaFirmaAlDelegado(FirmaMetodoAInvocar metodoAInvocar)
{
```

```
            this.metodoAInvocar += metodoAInvocar;
        }
```

VB

```
    Public Sub AgregaFirmaAlDelegado(metodoAInvocar As FirmaMetodoAInvocar)
        Me.metodoAInvocar = [Delegate].Combine(Me.metodoAInvocar, metodoAInvocar)
    End Sub
```

También se pueden quitar firmas del delegado de multidifusión y nuevamente esto depende del lenguaje usado:

- En C# se utiliza el operador -= para quitar la firma especificada. Se admite su uso aunque se quite una sola firma. También se puede utilizar el método Remove que se explica para VB, pero este el formato más difundido.
- En VB, se utiliza el método Remove que recibe dos argumentos, el primero con el delegado que contiene la lista de invocación, el segundo con el último delegado que se quitará de la lista. Este método retorna el delegado de multidifusión a utilizar.

Ejemplo

C#

```
public void QuitaFirmaAlDelegado(FirmaMetodoAInvocar metodoAInvocar)
{
       this.metodoAInvocar -= metodoAInvocar;
}
```

VB

```
Public Sub QuitaFirmaAlDelegado(metodoAInvocar As FirmaMetodoAInvocar)
       Me.metodoAInvocar = [Delegate].Remove(Me.metodoAInvocar, metodoAInvocar)
End Sub
```

Definición y uso de eventos

Un evento es la manera que tiene un objeto de notificar al código cliente que se suscriba al mismo cuando ocurre algo importante en él. El uso más habitual de los eventos es en las interfaces gráficas de usuario (GUI). Las clases que representan los controles en las interfaces gráficas del usuario lanzan los eventos cuando éste interactúa con el control. Por ejemplo, cuando el usuario hace clic en un botón, el objeto que define a dicho botón dispara un evento Click.

El uso de eventos no se limita a interfaces gráficas de usuario. Se puede usar eventos para representar los sucesos relacionados con una empresa, tales como un cambio en la situación financiera de un saldo de una determinada cuenta bancaria, la llegada de un mensaje a una cola, o la actualización de un registro en una base de datos.

Para utilizar eventos en una aplicación del Framework de .NET, se deben realizar las siguientes tareas:

1. Declarar un evento.
2. Lanzar el evento.
3. Manejar el evento.

La declaración de un evento

Los eventos se pueden declarar en una clase, estructura o interfaz. Para declarar un evento se utiliza la palabras clave `event` o `Event` (C# o VB). La declaración del evento debe especificar de firma de los métodos que lo manejen. Las técnicas siguientes se pueden utilizar para especificar la firma:

- Especificar la firma explícitamente como parte de la declaración del evento.
- Especificar la firma implícitamente mediante el uso de un delegado existente. El delegado indica la firma de los métodos del manejador de eventos.

El NET Framework define un patrón estándar para definir las firmas de los métodos manejadores de eventos, como sigue:

- El primer parámetro es de tipo System.Object y debe indicar el objeto que generó el evento.
- El segundo parámetro es del tipo System.EventArgs o una subclase de esta y debe utilizarse para transmitir información contextual sobre el evento (datos y métodos que asisten a su gestión por el manejador).

Si no se desea transmitir información contextual para un evento, se puede utilizar el delegado EventHandler. Este delegado especifica que los métodos del controlador (manejador) de eventos deben definir como parámetros un System.Object y un System.EventArgs.

Los ejemplos de código siguientes muestran cómo definir los eventos en una clase Cuenta para indicar cuando una cuenta de banco se mueve en crédito o déficit. Los ejemplos de código ilustran los siguientes puntos:

- La clase CuentaEventArgs es una clase para argumentos de evento personalizada que indica el saldo actual de la cuenta cuando se produce un evento.
- La clase Cuenta define dos eventos públicos que se denominan CuentaEnDeficit y CuentaConCredito. Ambos eventos se definen del tipo EventHandler, que especifica la firma de los métodos que son manejadores de eventos.

Ejemplo

C#

```
namespace eventos
{
    // Clase para argumento del evento personalizada.
    public class CuentaEventArgs : EventArgs
    {
        private double _balance;
        public CuentaEventArgs(double b)
```

```
        {
            _balance = b;
        }
        public double Balance
        {
            get { return _balance; }
        }
    }
}



namespace eventos
{
    // La clase Cuenta define dos eventos públicos
    class Cuenta
    {
        public event EventHandler CuentaEnDeficit;
        public event EventHandler CuentaConCredito;

        …
    }
}
```

VB

```
' Clase para argumento del evento personalizada.
Public Class CuentaEventArgs
    Inherits EventArgs
    Private _balance As Double
    Public Sub New(ByVal b As Double)
        _balance = b
    End Sub
    Public ReadOnly Property Balance() As Double
        Get
            Return _balance
        End Get
    End Property
End Class



' La clase Cuenta define dos eventos públicos
Public Class Cuenta

    Public Event CuentaEnDeficit As EventHandler
    Public Event CuentaConCredito As EventHandler

    …

End Class
```


Lanzar un evento

La sintaxis para lanzar un evento depende del lenguaje, si está utilizando C# o VB:

- Para generar un evento en C#, primero se debe comprobar si hay algún método registrado como manejador de evento para el mismo. Para hacerlo, comprobar si el evento no es nulo. Si este es el caso, se puede proceder a lanzar el evento utilizando su nombre, seguido de la lista de los argumentos entre paréntesis que se le desea pasar.
- Para lanzar un evento en VB, utilizar la palabra clave `RaiseEvent`, seguida por el nombre del evento junto con su lista de argumentos entre paréntesis que se le desea pasar.

Los ejemplos siguientes de código muestran cómo generar un evento en la clase Cuenta. El método Deposito, por ejemplo, comprueba si el saldo ha pasado a ser mayor que cero. En esta situación, el método crea un objeto del tipo CuentaEventArgs para guardar el saldo de la cuenta corriente y luego se lanza el evento CuentaConCredito.

Ejemplo

C#

```
namespace eventos
{
    // La clase Cuenta define dos eventos públicos
    class Cuenta
    {
        …

        private double _balance;
        private String _nombre;

        …

        public void Deposito(double cantidad)
        {
            _balance += cantidad;
            if (_balance > 0 && _balance <= cantidad)
            {
              // Se acaba de obtener crédito por balance positivo, de manera que
              // se lanza el evento CuentaConCredito. Se crea una copia del objeto
              // del evento para prevenir una condición de carrera (race condition).
                EventHandler manejador = CuentaConCredito;
                if (manejador != null)
                {
                    CuentaEventArgs args = new CuentaEventArgs(_balance);
                    manejador(this, args);
                }
            }
            else
            {
              // Se acaba de obtener crédito con balance negativo, de manera que
              // se lanza el evento CuentaEnDeficit. Se crea una copia del objeto
              // del evento para prevenir una condición de carrera (race condition).
                EventHandler manejador = CuentaEnDeficit;
                if (manejador != null)
                {
                    CuentaEventArgs args = new CuentaEventArgs(_balance);
                    manejador(this, args);
```

```
                }
            }
        }

        …
    }
}
```

VB

```
Public Class Cuenta

    …

    Private _balance As Double
    Private _nombre As String

    …

    Public Sub Deposito(ByVal cantidad As Double)
        _balance += cantidad
        If _balance > 0 AndAlso _balance <= cantidad Then
            ' Se acaba de obtener crédito por balance positivo, de manera que
            ' se lanza el evento CuentaConCredito.
            Dim args As New CuentaEventArgs(_balance)
            RaiseEvent CuentaConCredito(Me, args)
        Else
            ' Se acaba de obtener crédito con balance negativo, de manera que
            ' se lanza el evento CuentaEnDeficit.
            Dim args As New CuentaEventArgs(_balance)
            RaiseEvent CuentaEnDeficit(Me, args)
        End If
    End Sub
    …
End Class
```


Manejar eventos

Para gestionar un evento, definir mediante la firma que la declaración de evento especifica, un método manejador de eventos. A continuación, agregar el método de controlador de eventos para el mismo mediante el operador += o `AddHandler` (C# o VB). Se puede agregar más de un método manejador de eventos para el mismo evento. Cuando se produce el evento, el CLR invoca los métodos de controlador de eventos en el orden en que se agregaron. Para eliminar un método de la lista de métodos manejadores de eventos, utilizar el operador -= o `RemoveHandler` (C# o VB).

Los ejemplos de código siguientes muestran cómo controlar el evento CuentaConCredito en un objeto del tipo Cuenta. El método manejador de eventos OnCuentaConCredito recupera el saldo de la cuenta corriente del parámetro del tipo CuentaEventArgs y muestra el balance en la ventana de la consola.

Ejemplo

C#

```
namespace eventos
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear un objeto Cuenta.
            Cuenta _cuenta = new Cuenta("Juan Pérez");
            // Manejar el evento CuentaConCredito en el objeto Cuenta.
            _cuenta.CuentaConCredito += OnCuentaConCredito;
            _cuenta.CuentaEnDeficit += OnCuentaEnDeficit;
            // Depositar y retirar algo de dinero.
            _cuenta.Retiro(100);
            _cuenta.Deposito(50);
            _cuenta.Deposito(70);

            Console.ReadKey();
        }
        // Manejador del evento CuentaConCredito.
        static void OnCuentaConCredito(object sender, EventArgs args)
        {
            double balance = ((CuentaEventArgs)args).Balance;
            Console.WriteLine("Cuenta con crédito, el nuevo balance es: " +
balance);
        }

        // Manejador del evento CuentaEnDeficit.
        static void OnCuentaEnDeficit(object sender, EventArgs args)
        {
            double balance = ((CuentaEventArgs)args).Balance;
            Console.WriteLine("Cuenta conn déficit, el nuevo balance es: " +
balance);
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        ' Crear un objeto Cuenta.
        Dim _cuenta As New Cuenta("Jane")
        ' Manejar el evento CuentaConCredito en el objeto Cuenta.
        AddHandler _cuenta.CuentaConCredito, AddressOf OnCuentaConCredito
        AddHandler _cuenta.CuentaEnDeficit, AddressOf OnCuentaEnDeficit
        ' Depositar y retirar algo de dinero.
        _cuenta.Retiro(100)
        _cuenta.Deposito(50)
        _cuenta.Deposito(70)
        Console.ReadKey()
    End Sub
```

```
    ' Manejador del evento CuentaConCredito.
    Public Sub OnCuentaConCredito(ByVal sender As Object, _
    ByVal args As CuentaEventArgs)
        Dim balance As Double = args.Balance
        Console.WriteLine("Cuenta con crédito, el nuevo balance es: " + _
              balance.ToString)
    End Sub

    ' Manejador del evento CuentaEnDeficit.
    Public Sub OnCuentaEnDeficit(ByVal sender As Object, _
                                 ByVal args As CuentaEventArgs)
        Dim balance As Double = args.Balance
        Console.WriteLine("Cuenta conn déficit, el nuevo balance es: " + _
              balance.ToString)
    End Sub
End Module
```

Definiciones parciales

Es posible dividir la definición de una clase, estructura, interfaz o método en dos o más archivos de código fuente. Cada archivo de código fuente puede contener una parte de la definición estos, y todas las partes se combinan cuando se compila la aplicación.

Se debe tener cuidado al utilizar definiciones parciales porque pueden generar problemas de mantenimiento de código. Sin embargo, estas definiciones son una salida elegante que Microsoft ha encontrado al problema de la generación automática de código por parte de los diseñadores, como el de formularios (el más comúnmente utilizado, pero no el único). Al poder generar definiciones parciales, el programador no tiene que interaccionar con el código generado y puede adicionar aquello que crea conveniente para la ejecución de su aplicación, volver al diseñador, hacer modificaciones que generen nuevo código y estar seguro que no se alteró aquel que haya escrito específicamente. El correcto uso de los diseñadores se discutirá posteriormente en el capítulo de interfaces gráficas.

Clases parciales

Para dividir una definición de clase, se debe utilizar el modificador `partial` o `Partial` (C# o VB), como se muestra a continuación.

Ejemplo

C#

```
namespace parciales
{
    public partial class Empleado
    {
        private String nombre = "Juan Pérez";
        private double salario = 1500.00;

        public virtual String GetDetalles()
```

```
        {
            return "Nombre: " + nombre + "\nSalario: " + salario;
        }
        public String Nombre
        {
            get
            {
                return nombre;
            }
            set
            {
                nombre = value;
            }
        }
        public double Salario
        {
            get
            {
                return salario;
            }
            set
            {
                salario = value;
            }
        }
    }
}
```

VB

```
Partial Public Class Empleado
    Private _nombre As String = "Juan Pérez"
    Private _salario As Double = 1500.0

    Public Overridable Function GetDetalles() As String
        Return "Nombre: " + _nombre + Environment.NewLine + "Salario: " +
_salario.ToString
    End Function
    Public Property Nombre() As String
        Get
            Return _nombre
        End Get
        Set(ByVal value As String)
            _nombre = value
        End Set
    End Property
    Public Property Salario() As Double
        Get
            Return _salario
        End Get
        Set(ByVal value As Double)
            _salario = value
        End Set
    End Property
End Class
```

Como la clase Empleado fue declarada como parcial, implica que admite que exista otra definición parcial que se agregue a su definición en tiempo de compilación

C#

```
namespace parciales
{
    public partial class Empleado
    {
        public void Trabajar()
        {
            Console.WriteLine("El empleado está trabajando");
        }
    }
}
```

VB

```
Partial Public Class Empleado
    Public Sub Trabajar()
        Console.WriteLine("El empleado está trabajando")
    End Sub
End Class
```

Se debe pensar a cualquier definición parcial como que ***se completa la definición previa, al igual que se haría si se estuviese agregando código a la clase, estructura, interfaz o método que ya se hubiera definido en un mismo archivo***. Por lo tanto, cualquier error de definición en una situación como esa, será un error en la definición parcial.

Se debe tener en cuenta en las declaraciones parciales que:

- La palabra clave `partial` o `Partial` (C# o VB) indica que otras partes de la clase, estructura, interfaz o método se pueden definir en el mismo espacio de nombres.
- Todas las partes deben utilizar la palabra clave `partial` o `Partial` (C# o VB).
- Todas las partes deben estar disponibles en tiempo de compilación para formar la declaración final.
- Todas las partes deben tener la misma accesibilidad, ya sea `public`, `private`, etc. o `Public` o `Private` (C# o VB).
- Si alguna parte se declara abstracta, todo el tipo se considera abstracto.
- Si alguna parte se declara `sealed` o `NotInheritable` (C# o VB), todo el tipo se considera `sealed` o `NotInheritable` (C# o VB).
- Si alguna parte declara una superclase, todo el tipo hereda esa clase.
- Todas las partes que especifican una clase base deben concordar (es decir, no se deben cambiar las declaraciones de superclase de una definición parcial a otra), pero las partes que omiten una clase base heredan igualmente el tipo base que se declare en alguna de sus partes.
- Las partes pueden especificar diferentes interfaces, pero el tipo final implementa todas las interfaces mostradas por todas las declaraciones parciales.

- Cualquier miembro de clase, estructura, interfaz o método declarado en una definición parcial está disponible para todas las demás partes.
- El tipo final es la combinación de todas las partes en tiempo de compilación.

Métodos parciales

Una clase o estructura parcial puede contener un método parcial. Si una parte de la clase contiene la firma del método, se puede definir opcionalmente una implementación en la misma parte o en otra. Si no se proporciona la implementación, el método y todas las llamadas a él se quitan en tiempo de compilación.

Los métodos parciales permiten al implementador de una parte de una clase definir un método, de forma similar a un evento. El programador de la otra parte de la clase puede decidir si implementar el método o no. Si no se implementa el método, el compilador quita la firma del método y todas las llamadas a éste. Dichas llamadas al método, incluidos los resultados ocurridos a partir de la evaluación de los argumentos en las llamadas, no tienen ningún efecto en tiempo de ejecución. Por consiguiente, cualquier código en la clase parcial puede utilizar libremente un método parcial, aun cuando no se proporcione la implementación del mismo. No se producirá ningún error en tiempo de compilación o en tiempo de ejecución si se realizan llamadas al método y éste no está implementado.

Los métodos parciales son especialmente útiles como una manera de personalizar el código generado. Permiten reservar una firma de método, de modo que el código generado pueda llamar al mismo y el programador pueda decidir si lo implementa o no. De forma muy similar a las clases parciales, los métodos parciales permiten que el código creado por un generador de código y el creado por un programador humano puedan funcionar juntos sin costo en tiempo de ejecución.

Una declaración de método parcial consta de dos partes: la definición y la implementación. Éstas pueden estar en partes independientes de una clase parcial o en la misma. Si no existe ninguna declaración de implementación, el compilador quita la declaración de definición y todas las llamadas al método.

A continuación se realiza una modificación de una clase usada anteriormente, Cuenta, para incluir los manejadores de eventos como parte de una clase parcial. Notar que como los métodos parciales son implícitamente privados, los manejadores se definen como métodos parciales en la misma clase y la asignación de los manejadores de eventos se realiza en el constructor.

Ejemplo

C#

```
namespace metodosParciales
{
    // La clase Cuenta define dos eventos públicos
    public partial class Cuenta
    {
```

```
        public event EventHandler CuentaEnDeficit;
        public event EventHandler CuentaConCredito;

        private double _balance;
        private String _nombre;

        public Cuenta(String nombre)
        {
            this._nombre = nombre;

            // Manejar el evento CuentaConCredito en el objeto Cuenta.
            CuentaConCredito += OnCuentaConCredito;
            CuentaEnDeficit += OnCuentaEnDeficit;
        }

        partial void OnCuentaConCredito(object sender, EventArgs args);
        partial void OnCuentaEnDeficit(object sender, EventArgs args);

      …
    }
}
```

En otro archivo se define:

```
namespace metodosParciales
{
    public partial class Cuenta
    {
        // Manejador del evento CuentaConCredito.
        partial void OnCuentaConCredito(object sender, EventArgs args)
        {
            double balance = ((CuentaEventArgs)args).Balance;
            Console.WriteLine("Cuenta con crédito, el nuevo balance es: " +
                    balance);
        }

        // Manejador del evento CuentaEnDeficit.
        partial void OnCuentaEnDeficit(object sender, EventArgs args)
        {
            double balance = ((CuentaEventArgs)args).Balance;
            Console.WriteLine("Cuenta conn déficit, el nuevo balance es: " +
                    balance);
        }
    }
}
```

VB

```
Partial Public Class Cuenta

    Public Event CuentaEnDeficit As EventHandler
    Public Event CuentaConCredito As EventHandler

    Private _balance As Double
    Private _nombre As String
```

```
    Public Sub New(nombre As String)
        Me._nombre = nombre

        ' Manejar el evento CuentaConCredito en el objeto Cuenta.
        AddHandler CuentaConCredito, AddressOf OnCuentaConCredito
        AddHandler CuentaEnDeficit, AddressOf OnCuentaEnDeficit

    End Sub

    Partial Private Sub OnCuentaConCredito(ByVal sender As Object, _
                ByVal args As CuentaEventArgs)

    End Sub
    Partial Private Sub OnCuentaEnDeficit(ByVal sender As Object, _
                              ByVal args As CuentaEventArgs)

    …
End Class
```

En otro archivo se define:

```
Partial Public Class Cuenta
    ' Manejador del evento CuentaConCredito.
    Private Sub OnCuentaConCredito(ByVal sender As Object, _
    ByVal args As CuentaEventArgs)
        Dim balance As Double = args.Balance
        Console.WriteLine("Cuenta con crédito, el nuevo balance es: " + _
              balance.ToString)
    End Sub

    ' Manejador del evento CuentaEnDeficit.
    Private Sub OnCuentaEnDeficit(ByVal sender As Object, _
                                  ByVal args As CuentaEventArgs)
        Dim balance As Double = args.Balance
        Console.WriteLine("Cuenta conn déficit, el nuevo balance es: " + _
              balance.ToString)
    End Sub
End Class
```

Para utilizar métodos parciales, tener en cuenta que:

- Las declaraciones de método parciales deben comenzar con la palabra clave `partial` o `Partial` (C# o VB), y el método debe retornar `void` o ser una `Sub` (C# o VB).
- Los métodos parciales pueden tener parámetros `ref`, pero ***no*** **`out`**.
- Los métodos parciales son implícitamente privados y, por consiguiente, no pueden ser virtual ni tener modificadores de acceso (más aún, VB exige que se los declare privados explícitamente).
- Los métodos parciales pueden tener modificadores `static` o `Shared` (C# o VB).
- Se puede crear un delegado de un método parcial que se ha definido e implementado, pero no de un método parcial que solo se ha definido. Este es análogo para los eventos.

Características especiales de las propiedades

Una propiedad tiene un tipo y un nombre al igual que una variable de instancia. Sin embargo, la lógica de una propiedad se define mediante los descriptores de acceso `get` y `set` o `Get` y `Set` (C# o VB).

El descriptor de acceso `get` o `Get` (C# o VB), como un método, puede incluir cualquier código, sin embargo, debe devolver un objeto del tipo especificado por la propiedad o una excepción. El descriptor de acceso `set` o `Set` (C# o VB) no tiene que realizar ninguna función, aunque normalmente, actualiza un atributo privado para realizar alguna operación basada en el valor pasado a la propiedad. No se especifica un parámetro para el descriptor de acceso `set` o `Set` (C# o VB), ya que éste siempre toma un parámetro del mismo tipo que el expuesto por la propiedad. Se puede acceder al objeto que se pasa como parámetro a un descriptor de acceso `set` o `Set` (C# o VB) con la palabra clave `value`.

En el ejemplo de código siguiente se muestra cómo definir una propiedad simple que proporciona acceso a una variable de instancia privada. La palabra clave `get` o `Get` (C# o VB) introduce un bloque de código que define lo que se ejecuta cuando cierto código cliente lee el valor que almacena la propiedad. La palabra clave `set` o `Set` (C# o VB) define el bloque de código de la lógica que se ejecuta cuando se asigna un valor a la propiedad

Ejemplo

C#

```
namespace propiedades
{
    class Propiedad
    {
        private string cadena;

        public string Cadena
        {
            get { return cadena; }
            set { cadena = value; }
        }
    }
}
```

VB

```
Public Class Propiedad
    Private _cadena As String
    Public Property Cadena() As String
        Get
            Return _cadena
        End Get
        Set(ByVal value As String)
            _cadena = value
        End Set
    End Property
```

```
End Class
```

Propiedades automáticas

Cuando se desarrolla un nuevo tipo (por ejemplo, con una clase), se puede querer incluir un atributo que se desee exponer al código cliente. Si no hay ningún tratamiento adicional o es necesaria una validación en este atributo, puede ser tentador simplemente exponerlo públicamente lugar de añadir una propiedad para facilitar el acceso al mismo. Obviamente, el primer principio que esto viola abiertamente es el de ocultamiento de la información. Sin embargo, el planteo se focaliza en las consecuencias directas en el cósdigo sin tener en cuenta dicho principio.

En este caso, la exposición de un atributo no parece ser un problema. Sin embargo, recordar que no se puede agregar código para evitar valores no válidos en el mismo, pero si se puede en una propiedad. Ya sea que se necesite agregar validación u otra lógica a una propiedad cuando se desarrolla originalmente un tipo, no significa que siempre sea así. Los requisitos del tipo pueden cambiar con el tiempo de vida de la aplicación.

Desde la perspectiva del desarrollador, utilizar una propiedad es exactamente lo mismo que usar una variable, sin embargo, esto no es cierto para el compilador. El compilador convierte el código que accede a una propiedad en una llamada al método de descriptor de acceso `get` o `Get` (C# o VB), y de manera similar, convierte la escritura en una propiedad en una llamada al método de acceso `set` o `Set` (C# o VB). Esto tiene implicaciones para las aplicaciones existentes si es necesario convertir un atributo a una propiedad en una fecha posterior, cualquier código cliente que utiliza el tipo con el valor expuesto como una variable de instancia debe ser recompilado con los datos que a partir de ese momento necesitan ser expuestos a través de una propiedad. Si este tipo está en un ensamblado utilizado por un número de aplicaciones, puede que se tenga que reconstruir y volver a desplegar una gran cantidad de instalaciones.

Se puede evitar este trabajo extra con sólo exponer los datos a través de una propiedad cuando originalmente se desarrolla el tipo. Cualquier cambio futuro en el tipo se puede hacer sin necesidad de recompilar las aplicaciones que consumen su tipo si se utiliza esta manera.
Dónde se puede exponer un atributo y exista la tentación de hacer simplemente una variable de instancia pública en lugar de escribir una propiedad para obtener y establecer el valor del atributo, se pueden utilizar las propiedades automáticas.

Las propiedades automáticas proporcionan una línea simple de sintaxis que convierte una variable de instancia en una propiedad. Para utilizar las propiedades automáticas, sólo se tiene que añadir el código apropiado según el lenguaje, como muestra el siguiente ejemplo.

Ejemplo

C#

```
namespace propiedades
{
    class Automaticas
    {
        public string Nombre { get; set; }
        public string Apellido { get; set; }
    }
}
```

VB

```
Public Class Automaticas
    Public Property Nombre As String
    Public Property Apellido As String
End Class
```

Creación de objetos usando propiedades

Ya se ha visto cómo utilizar un constructor para crear instancias de un objeto e inicializar sus atributos. Se pueden declarar varios constructores, con diferentes firmas, para permitir a otros desarrolladores crear varias combinaciones de atributos como opciones diferentes de creación de un objeto con los valores adecuados, sin embargo, este enfoque es problemático si se tiene más de una variable o propiedad del mismo tipo.

En el siguiente ejemplo de código se muestra una clase simple, con varios constructores.

Ejemplo

C#

```
namespace propiedades
{
    public class Empleado
    {
        private string nombre;
        private double salario;
        private string departamento;

        public Empleado(string nombre)
        {
            this.nombre = nombre;
        }

        public Empleado(double salario)
        {
            this.salario = salario;
        }

        public Empleado(string departamento)
        {
            this.departamento = departamento;
```

```
        }

        public Empleado(string nombre, double salario, string departamento)
        {
            this.nombre = nombre;
            this.salario = salario;
            this.departamento = departamento;
        }

        …
    }
}
```

VB

```
Public Class Empleado
    Private _nombre As String
    Private _salario As Double
    Private _departamento As String

    Public Sub New()

    End Sub

    Public Sub New(nombre As String)
        Me.Nombre = nombre
    End Sub

    Public Sub New(salario As Double)
        Me.Salario = salario
    End Sub

    Public Sub New(departamento As String)
        Me.Departamento = departamento
    End Sub

    Public Sub New(nombre As String, salario As Double, departamento As String)
        Me.Nombre = nombre
        Me.Salario = salario
        Me.Departamento = departamento
    End Sub

    …

End Class
```


La intención de los constructores es permitir al código cliente especificar un valor para el nombre del empleado, nombre del departamento, o el sueldo, cuando se crea un nuevo objeto Empleado. Sin embargo, este código no se compilará porque el compilador no puede distinguir entre los dos constructores que toman un único parámetro como cadena. Si intenta crear una instancia de un objeto Empleado utilizando el código que se muestra en el ejemplo anterior, el compilador no sabe qué constructor de usar.

Se puede resolver este problema mediante el uso de propiedades para inicializar el objeto cuando se crea la instancia. Esta sintaxis se conoce como un *inicializador*. Con un inicializador de objeto, se crea un nuevo objeto mediante el uso de un constructor, pero se especifican los valores a asignar a las propiedades después que el constructor se ha ejecutado mediante el uso de asignaciones por medio de pares nombre/valor separados por comas. Los pares nombre/valor van encerrados entre llaves y la sintaxis varía levemente en VB por el uso de la palabra clave `With` y un punto antecediendo al nombre.

En el ejemplo de código siguiente se muestra cómo definir una clase que admite inicializadores de objeto y cómo crear objetos con ellos. Notar que si se usa el constructor por defecto no es necesario incluir los paréntesis y que se pueden usar otros constructores definidos a la vez que se asignan valores a las propiedades.

Ejemplo

C#

```
namespace propiedades
{
    public class Empleado
    {
        public string Nombre { get; set; }
        public double Salario { get; set; }
        public string Departamento { get; set; }

        public Empleado(string nombre, double salario, string departamento)
        {
            Nombre = nombre;
            Salario = salario;
            Departamento = departamento;
        }

        public Empleado(string nombre, double salario)
        {
            Nombre = nombre;
            Salario = salario;
        }

        public Empleado()
        {
        }

        public virtual String GetDetalles()
        {
            return "Nombre: " + Nombre + "\nSalario: " + Salario +
              "\nDepartamento: " + Departamento;
        }
    }
}
```

La declaración de los objetos es la siguiente:

```
            Empleado e1 = new Empleado
            {
                Nombre = "Juan Pérez",
                Departamento = "Ventas",
                Salario = 1000
            };

            Empleado e2 = new Empleado("Oscar Toma", 20000)
            {
                Departamento = "Ingeniería"
            };
```

VB

```
Public Class Empleado
    Public Property Nombre As String
    Public Property Salario As Double
    Public Property Departamento As String

    Public Sub New()

    End Sub

    Public Sub New(nombre As String, salario As Double, departamento As String)
        Me.Nombre = nombre
        Me.Salario = salario
        Me.Departamento = departamento
    End Sub

    Public Sub New(nombre As String, salario As Double)
        Me.Nombre = nombre
        Me.Salario = salario
    End Sub

    Public Overridable Function GetDetalles() As String
        Return "Nombre: " + Nombre + Environment.NewLine + "Salario: " + _
            Salario.ToString + Environment.NewLine + "Departamento: " + _
            Departamento
    End Function
End Class
```

La declaración de los objetos es la siguiente:

```
        Dim e1 As New Empleado With
            {
                .Nombre = "Juan Pérez",
                .Departamento = "Ventas",
                .Salario = 1000
            }

        Dim e2 As New Empleado("Oscar Toma", 20000) With
            {
                .Departamento = "Ventas"
            }
```

Indexadores y propiedades por defecto

Los tipos, incluidas las interfaces, puede tener una propiedad predeterminada. Las propiedades predeterminadas se utilizan como una notación abreviada para el acceso a los elementos de un vector o colección de objetos contenidos dentro de otro objeto.

En C#, las propiedades predeterminadas con índices se llaman indexadores y no existen de otra forma que no sean con índices. Por otro lado, VB exige que se le ponga un nombre a la propiedad por defecto y el utilizado en estos casos es Item. Hay una sutil diferencia en la forma en éstos se declaran en C# y VB. Como colecciones se verá en un capítulo dedicado a ellas, la siguiente explicación se dará para el caso de un vector, pero es análoga para colecciones.

Escribir un indexador es un cruce entre escribir una propiedad y el uso de un vector. Utilizar la sintaxis de las propiedades para especificar los descriptores de acceso `get` y `set` o `Get` y `Set` (C# o VB), pero el nombre del indexador siempre es `this` en C# y el nombre por defecto en VB es Item. Se especifican los tipos y nombres de parámetros mediante el uso como en los vectores con notación entre corchetes para C# y paréntesis para VB.

Al igual que una propiedad, un indexador también puede ser de sólo lectura (sólo tiene un descriptor de acceso `get` en C# y la palabra clave `ReadOnly` en VB) o sólo escritura (sólo tiene un descriptor de acceso `set` y la palabra clave `WriteOnly` en VB).

Los parámetros pasados a un indexador están destinados únicamente a ser utilizado para localizar el elemento de datos para establecer u obtener el valor almacenado en el vector o colección interno. En el descriptor de acceso `get` o `Get` (C# o VB), retorna el elemento encontrado en este lugar, y en el descriptor de acceso `set` o `Set` (C# o VB), almacena el valor especificado por el parámetro `value` o Value (C# o VB) en ese lugar.

Ejemplo

C#

```
namespace indexadores
{
    class PropiedadesPorDefecto
    {
        private int[] vec = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

        // Implementación de propiedad por defecto
        // <modificador> <tipo retornado> this[Object <nombre>]
        public int this[int indice]
        {
            get
            {
                return vec[indice];
            }

            set
```

```
            {
                vec[indice] = value;
            }
        }
    }
}
```

VB

```
Public Class PropiedadesPorDefecto
    Private vec() As Integer = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}

    ' Implementación de propiedad por defecto
    ' Default <modificador> Property <nombre>(nombre as Object) As <tipo retornado>
    Default Public Property Item(ByVal I As Integer) As Integer
        Get
            Return vec(I)
        End Get

        Set(ByVal Value As Integer)
            vec(I) = Value
        End Set
    End Property
End Class
```

Para utilizar un indexador, la sintaxis similar a la de un vector, sin embargo, hay varias diferencias importantes entre ambos.

- **Los subíndices**: Cuando se usa un vector, se accede a los miembros del mimsmo un subíndice numérico. Por ejemplo, se puede acceder al quinto elemento de un vector utilizando una sintaxis similar a vector[4] (suponiendo un índice basado en cero). Con los vectores, sólo se pueden usar subíndices numéricos. Un indixador permite una mayor flexibilidad, ya que puede utilizar subíndices no numéricos.
- **Sobrecargar un indexador**: No puede sobrecargar un vector, la implementación se define en tiempo de ejecución, y todas las clases que heredan de la que lo defina no puede cambiar el comportamiento del mismo. Sin embargo, se tiene el control completo sobre el comportamiento de un indexador, y las clases que heredan de la que lo define pueden anular el indexador y proporcionar su propia implementación.
- **El uso de un indexador como parámetro**: Las dos anteriores diferencias son los beneficios de usar un indexador en lugar de un vector. Ambas son ciertas, porque cuando se utiliza un indexador, efectivamente se llama a un método en la clase (aunque esto es manejado por el compilador). Cuando se llama a un método que toma un parámetro `ref` u `out`, se debe pasar un puntero a una ubicación de memoria que utiliza el método. Los elementos de un vector se pueden asignar directamente a dichas ubicaciones de memoria, por lo que se los puede utilizar como un argumento a un método que toma un parámetro `ref` u `out`. Los indexadores no se asignan directamente a lugares de memoria, así que no se pueden

usar un indexador como parámetro `ref` u `out`, aunque se pueden pasar como parámetros por valor.

Sobrecarga de operadores

Una característica especial de las clases es permitir que los objetos que se declaren de su tipo puedan definir el comportamiento de ciertos operadores cuando se los aplica en una expresión que los incluya.

El Framework de .Net incluye varios operadores que permiten realizar las operaciones más comunes con objetos. Se puede utilizar estos operadores para construir expresiones en las cuales se predefine el comportamiento del operador dentro de dicha expresión y dicho comportamiento exacto para cada uno de los operadores dependerá del tipo del objeto que realiza la operación.

Un operador es un método especial que tiene un conjunto de parámetros y retorna un valor. Cuando se invoca a un operador, los operandos se transfieren como parámetros para dicho método, y el valor retornado por el mismo se comporta como el resultado del operador. Cuando se sobrecarga un operador, se proporciona una propia implementación de dicho método.

Los lenguajes C# y VB definen tres categorías de operadores que se pueden sobrecargar:

- Los operadores unarios. Cuando se sobrecarga estos operadores, se especifica un único parámetro, que debe ser del mismo tipo que la clase que define el operador.
- Los operadores binarios. Cuando se sobrecarga estos operadores, se especifican dos parámetros, por lo menos uno de los cuales debe ser del mismo tipo que la clase que define el operador.
- Los operadores de conversión. Puede utilizar estos operadores para cambiar los datos de un tipo a otro. Cuando se sobrecarga a estos operadores, se especifica un único parámetro que contiene los datos que desea convertir. Estos datos pueden ser de cualquier tipo válido.

No puede sobrecargar todos los operadores definidos. Las siguientes tablas resumen los operadores que se pueden o no puede sobrecargar.

C#

Operadores	Capacidad para ser sobrecargado
+, -, !, ~, ++, --, true, false	Estos operadores unarios se pueden sobrecargar.
+, -, *, /, %, &, |, ^, <<, >>	Estos operadores binarios se pueden sobrecargar.
==, !=, <, >, <=, >=	Los operadores de comparación se pueden sobrecargar.
&&, ||	Los operadores lógicos condicionales no se puede sobrecargar, pero se evalúan utilizando & y |, que se puede sobrecargar.
[]	El operador de indización de matriz no se puede sobrecargar, pero se pueden definir indizadores.

Operadores	Capacidad para ser sobrecargado
()	El operador de conversión no se puede sobrecargar, pero se pueden definir nuevos operadores de conversión, como se describe más adelante en este capítulo.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Los operadores de asignación no se puede sobrecargar, pero + =, por ejemplo, se evalúa usando +, la cual se puede sobrecargar.
=, ., ?:, ->, new, is, sizeof, typeof	Estos operadores no se pueden sobrecargar.

VB

Operadores	Capacidad para ser sobrecargado
+, -, Not, IsTrue, IsFalse, CType	Estos operadores unarios se pueden sobrecargar.
+, -, *, /, \, Like, Mod, And, Or, Xor, ^, <<, >>	Estos operadores binarios se pueden sobrecargar.
=, <>, <, >, <=, >=	Los operadores de comparación se pueden sobrecargar.
AndAlso, OrElse	Los operadores lógicos condicionales no se puede sobrecargar, pero se evalúan utilizando And y Or, que se puede sobrecargar.
()	El operador de indexación de matriz no se puede sobrecargar.
AsType	El operador de conversión no se puede sobrecargar, pero se pueden definir nuevos operadores de conversión, como se describe más adelante en este capítulo.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Los operadores de asignación no se pueden sobrecargar, pero +=, por ejemplo, se evalúa usando +, el cual se puede sobrecargar.
New, TypeOf... Is, Is, IsNot, AddressOf y GetType	Estos operadores no se pueden sobrecargar.

Sobrecargando un operador

Para definir un comportamiento propio para un operador, se debe sobrecargar el operador seleccionado. Se utiliza una sintaxis como la de un método con un tipo retornado y parámetros, pero el nombre del método es la palabra clave `operator` u `Operator` (C# o VB) junto con el símbolo del operador que está sobrecargando. Por ejemplo, para sobrecargar el operador +, se define un método llamado operador `operator` + u `Operator` + (C# o VB).

El siguiente ejemplo de código muestra como se define un operador binario + para sumar dos instancias de un objeto.

Ejemplo

C#

```
namespace operadores
{
    class Complejo
    {
        private int real;
        private int imaginario;

        public Complejo(int real, int imaginario)
        {
            this.real = real;
            this.imaginario = imaginario;
        }

        // Declarar el operador a sobrecargar (+), los tipos
        // que se van a sumar (dos objetos del tipo Complejo)
        // y el tipo retornado (un objeto del tipo Complejo)
        public static Complejo operator +(Complejo c1, Complejo c2)
        {
            return new Complejo(c1.real + c2.real, c1.imaginario + c2.imaginario);
        }
        public string GetComplejo()
        {
            return (real + " + " +  imaginario + "i");
        }
    }
}
```

VB

```
Public Class Complejo
    Private real As Integer
    Private imaginario As Integer

    Public Sub New(real As Integer, imaginario As Integer)
        Me.real = real
        Me.imaginario = imaginario
    End Sub

    ' Declarar el operador a sobrecargar (+), los tipos
    ' que se van a sumar (dos objetos del tipo Complejo)
    ' y el tipo retornado (un objeto del tipo Complejo)

    Public Shared Operator +(c1 As Complejo, c2 As Complejo) As Complejo
        Return New Complejo(c1.real + c2.real, c1.imaginario + c2.imaginario)
    End Operator

    Public Function GetComplejo() As String
        Return (real.ToString + " + " + imaginario.ToString + "i")
    End Function
End Class
```


Tener en cuenta los siguientes puntos sobre el método del operador + como ejemplo:

- Todos los operadores deben ser públicos.
- Todos los operadores deben ser estáticos. Los operadores nunca son polimórficos y no se puede usar el modificador `virtual`, `override`, `abstract` y `sealed` u `Overridable`, `Overrides`, `MustOverride` y `NotOverridable` (C# o VB).

Consejo: Cuando se declara una funcionalidad muy particular (operadores), es útil adoptar una convención de nomenclatura para los parámetros. Por ejemplo, los desarrolladores a menudo usan diferencia el lado izquierdo y el derecho de la operación para los operadores binarios.

Cuando se utiliza el operador + entre dos expresiones de tipo Complejo, el compilador convierte automáticamente el código para llamar al método del operador + definido dentro de la clase.

Hay una regla final que debe seguir cuando se declara un operador: al menos uno de los parámetros siempre deben ser del tipo que contiene la definición del operador. Tomando el ejemplo anterior, el método para el operador + de la clase Complejo sigue la regla que uno de los parámetros, c1 o c2, debe ser un objeto del tipo Complejo. En este ejemplo, ambos parámetros lo son y por lo tanto aplican la regla.

Los operadores siguen las reglas habituales sobrecarga, y puede sobrecargar un operador tantas veces como desee en una clase, siempre y cuando el compilador pueda distinguir entre cada prototipo sobrecargado (la firma debe ser única en la clase). Por ejemplo, se puede definir una implementación adicional del método para el operador + que sume números de punto flotante.

Restricciones en la sobrecarga de operadores

Cuando se sobrecarga un operador, se puede controlar completamente cómo se realiza una operación, sin embargo, hay algunas reglas que se aplican a los operadores que no se pueden cambiar:

- No se puede cambiar la precedencia o asociatividad de un operador. La prioridad y la asociación se basa en el símbolo del operador (por ejemplo, +) y no en el tipo (por ejemplo, `int` o `Integer` –C# o VB) que está siendo utilizado por el símbolo del operador. Por lo tanto, la expresión a + b * c es siempre lo mismo que a + (b * c), independientemente de los tipos a, b, y c.
- No se puede cambiar la multiplicidad (el número de operandos) de un operador. Por ejemplo, * (el símbolo de multiplicación) es un operador binario (tiene dos operandos). Si se declara un operador * para un tipo propio, debe ser un operador binario. Del mismo modo, ++ es un operador unario (lleva un operando), si se declara ++ en un tipo, tiene que ser una operación unaria.
- No se pueden inventar nuevos símbolos de operador. Por ejemplo, no se puede crear un nuevo símbolo como operador, tal como **, para elevar un número a la potencia de otro número. Si tiene que realizar una operación para la que no hay ningún operador, se debe crear un método en su lugar.

- No se puede cambiar el significado de los operadores cuando se aplica a los tipos enteros. Por ejemplo, la expresión 1 + 2 tiene un significado predefinido, y no se puede anular este sentido. De hecho, cuando se define un operador en un tipo, al menos uno de los operandos para la operación debe ser del mismo tipo que lo contiene, por lo que no se puede definir una operación en la que todos los operandos son tipos predefinidos.

Además, se deben implementar los operadores de comparación por pares. Por ejemplo, si sobrecarga el operador "mayor que", también se debe sobrecargar el operador "menor que". Si sobrecarga el operador de "igualdad", también se debe sobrecargar el operador "distinto que".

La clase Object

La clase Object es superclase de las cadenas de herencia de todas las clases en el Framework de .Net, o sea, todas heredan de Object. Esto permite a los programas poder declarar referencias a Object y asignar cualquier tipo de objeto a la misma.

Aunque no se incluya en la declaración de una clase, implícitamente se deriva como si estuviese declarado.

Ejemplo

C#

```
namespace objeto
{
    class Empleado
    {
    }
}
```

VB

```
Public Class Empleado

End Class
```

Es lo mismo que:

C#

```
namespace objeto
{
    class Empleado: Object
    {
    }
}
```

VB

```
Public Class Empleado
    Inherits Object

End Class
```

Como los lenguajes permiten sólo asignar tipos de una subclase a uno de una superclase por conversiones explícitas de tipos (casting), esto habilita a las declaraciones de referencias de tipo Object actuar como "puentes" en las asignaciones (se convierte cualquier clase a Object y luego se puede convertir nuevamente a su tipo de referencia original porque todas las clases "heredan" de Object). Esto da gran flexibilidad de programación, como por ejemplo, crear vectores de tipo Object y asignar cualquier clase a ellos. *Sin embargo, se debe ser muy cuidadoso de no cometer errores al convertir nuevamente el objeto a su referencia original*.

Para aclarar detalles importantes de la clase Object, se necesita avanzar con algunos temas. Por lo tanto sólo se presentará una introducción al mismo.

Comparación de Objetos

El operador == o = (C# o VB) determina si dos referencias son iguales una a la otra, ya que los objetos se crean en el heap y en las variables del stack sólo se almacena la referencia al lugar de memoria en el que se encuentran. Por lo tanto queda por resolver todavía el problema de cómo comparar objetos.

Otra gran ventaja que se obtiene a partir que todas las clases heredan de Object es la de compartir sus métodos públicos. Basándose en esto, se puede resolver un problema muy común de la POO, el cual es responder a la pregunta ¿cuándo dos objetos son iguales? La respuesta es simple de concepto, pero complicada para implementarla en la codificación.

> *Dos objetos son iguales cuando sus atributos significativos lo son.*

Queda claro que el programador (y diseñador) deberá indicar la forma de comparar dos objetos ya que él decide que atributos son significativos para considerar a dos objetos iguales. Sin embargo resta encontrar el lugar para que esto se realice. En este punto llega al auxilio el hecho que Object es superclase de todas las clases en .Net, por lo tanto se puede utilizar un método público en ella para realizar la comparación. Sin embargo, luego cada programador deberá sobrescribir dicho método poniendo en su interior la funcionalidad adecuada que sea significativa en su clase para determinar cuando los objetos del tipo de la clase son iguales.

Por lo tanto, esto brinda la ventaja que el programador decida cuando dos objetos son iguales, pero en contrapartida le delega la responsabilidad que realice la comparación. El método Equals() brinda una manera de determinar si dos objetos son iguales a través de comparar sus atributos, ya que es un método de Object y se puede sobrescribir. Esto implica es el encargado de cumplir la acción antes descrita.

La implementación de Equals() en Object utiliza ==, por eso se debe sobrescribir para hacer algo distinto a comparar referencias. Si se sobrescribe Equals, se deberá sobrescribir GetHashCode() porque se utilizan en conjunto para comparar e identificar unívocamente a un objeto (esto se comprenderá mejor cuando se explique la utilización de colecciones).

En general, si se remplaza uno de estos métodos, es necesario remplazar ambos, ya que existen importantes relaciones entre ellos que deben ser mantenidas. En particular, si dos objetos son iguales de acuerdo con el método Equals, deben retornar el mismo valor GetHashCode (aunque la inversa no es cierta en general).

La forma en la cual Equals determina la igualdad de dos objetos se deja para el implementador, porque la definición de lo que es igual para dos objetos del mismo tipo es parte del trabajo de diseño para esa clase. La implementación por defecto, la que está escrita como código en Object, sólo compara la igualdad de ambas referencias.

Bajo esta implementación por defecto, dos referencias sólo son iguales si se refieren exactamente al mismo objeto.

Sin embargo, se debe cumplir con ciertos requisitos para que los programas funcionen correctamente a las que se denominan relaciones de equivalencia. Éstas se presentan en la siguiente tabla:

Relaciones de Equivalencia de Equals	
Reflexiva	Para todo valor de referencia x, x.Equals(x) debe retornar `true` o `True` (C# o VB)
Simétrica	Para todo valor de referencias x e y, x.Equals(y) debe retornar `true` o `True` (C# o VB) si también lo hace y.Equals(x)
Transitiva	Para todo valor de referencias x, y, z, si x.Equals(y) retorna `true` o `True` (C# o VB) y también lo hace y.Equals(z), entonces x.Equals(z) debe retornar `true` o `True` (C# o VB)
Consitente	Para todo valor de referencias x e y, múltiples invocaciones retornaran consistentemente verdadero o falso y nunca se alternará entre los resultados
No nulificable	Para todo valor de referencia x, x.Equals(`null`) o x.Equals(`Nothing`) debe retornar siempre `false` o `False` (C# o VB)

A continuación, se presenta la comparación de objetos en una clase utilizada en el módulo anterior a la que se le sobrescribe el método Equals():

Ejemplo

C#

```
namespace objeto
{
    class Empleado
    {
        private String nombre;
        private const double SALARIO_BASE = 1500.00;
        private double salario = 1500.00;
```

```
private Fecha fechaNacimiento;

public Empleado(String nombre, double salario, Fecha fDN)
{
    this.nombre = nombre;
    this.salario = salario;
    this.fechaNacimiento = fDN;
}
public Empleado(String nombre, double salario) :
    this(nombre, salario, null)
{
}

public Empleado(String nombre, Fecha fDN) :
    this(nombre, SALARIO_BASE, fDN)
{
}

public Empleado(String nombre) :
    this(nombre, SALARIO_BASE)
{
}

public override int GetHashCode()
{
    const int primo = 31;
    int resultado = 1;
    resultado = primo * resultado
            + ((fechaNacimiento == null) ? 0 :
                    fechaNacimiento.GetHashCode());
    resultado = primo * resultado + ((nombre == null) ? 0 :
                    nombre.GetHashCode());
    long temp;
    // Convertir a bits los puntos flotante para evitar errores
    // de redondeo en el resultado
    temp = BitConverter.DoubleToInt64Bits(salario);
    resultado = primo * resultado + (int)(temp ^ (temp >> 32));
    return resultado;
}

public override bool Equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (GetType() != obj.GetType())
        return false;
    Empleado otro = (Empleado)obj;
    if (fechaNacimiento == null)
    {
        if (otro.fechaNacimiento != null)
            return false;
    }
    else if (!fechaNacimiento.Equals(otro.fechaNacimiento))
```

```
                return false;
            if (nombre == null)
            {
                if (otro.nombre != null)
                    return false;
            }
            else if (!nombre.Equals(otro.nombre))
                return false;
        // Convertir a bits los puntos flotante para evitar errores
        // de redondeo en el resultado
            if (BitConverter.DoubleToInt64Bits(salario) !=
                    BitConverter.DoubleToInt64Bits(otro.salario))
                return false;
            return true;
        }
    }
}
```

VB

```
Public Class Empleado
    Private nombre As String
    Private fechaNacimiento As Fecha
    Private salario As Double
    Private Const SALARIO_BASE As Double = 15000.0

    ' Constructor
    Public Sub New(ByVal nombre As String, ByVal fDN As Fecha, ByVal salario As
Double)
        Me.nombre = nombre
        Me.fechaNacimiento = fDN
        Me.salario = salario
    End Sub

    Public Sub New(ByVal nombre As String, ByVal salario As Double)
        Me.New(nombre, Nothing, salario)
    End Sub

    Public Sub New(ByVal nombre As String, ByVal fDN As Fecha)
        Me.New(nombre, fDN, SALARIO_BASE)
    End Sub

    Public Sub New(ByVal nombre As String)
        Me.New(nombre, SALARIO_BASE)
    End Sub

    Public Overrides Function GetHashCode() As Integer
        Const primo As Integer = 31
        Dim resultado As Integer = 1
        Dim valor As Integer = IIf(fechaNacimiento Is Nothing, 0,
             fechaNacimiento.GetHashCode())
        resultado = primo * resultado + valor
        Dim valor2 As Integer = IIf((nombre Is Nothing), 0, nombre.GetHashCode())
        resultado = primo * resultado + valor2
        Dim temp As Long
        ' Convertir a bits los puntos flotante para evitar errores
```

```
        ' de redondeo en el resultado
        temp = BitConverter.DoubleToInt64Bits(salario)
        resultado = primo * resultado + (temp ^ (temp >> 32))
        Return resultado
    End Function

    Public Overrides Function Equals(obj As Object) As Boolean
        If Me Is obj Then
            Return True
        End If

        If obj Is Nothing Then
            Return False
        End If
        If Me.GetType() <> obj.GetType() Then
            Return False
        End If

        Dim otro As Empleado = CType(obj, Empleado)
        If fechaNacimiento Is Nothing Then
            If Not otro.fechaNacimiento Is Nothing Then
                Return False
            End If
        ElseIf Not fechaNacimiento.Equals(otro.fechaNacimiento) Then
            Return False
        End If
        If nombre Is Nothing Then
            If Not otro.nombre Is Nothing Then
                Return False
            End If
        ElseIf Not nombre.Equals(otro.nombre) Then
            Return False
        End If
        ' Convertir a bits los puntos flotante para evitar errores
        ' de redondeo en el resultado
        If BitConverter.DoubleToInt64Bits(salario) <> _
                BitConverter.DoubleToInt64Bits(otro.salario) Then
            Return False
        End If
        Return True
    End Function
End Class
```


Se puede comprobar el uso de las relaciones de equivalencia en el siguiente gráfico

C#

```
public override bool Equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (GetType() != obj.GetType())
        return false;
    Empleado otro = (Empleado)obj;
    if (fechaNacimiento == null)
    {
        if (otro.fechaNacimiento != null)
            return false;
    }
    else if (!fechaNacimiento.Equals(otro.fechaNacimiento))
        return false;
    if (nombre == null)
    {
        if (otro.nombre != null)
            return false;
    }
    else if (!nombre.Equals(otro.nombre))
        return false;
    if (BitConverter.DoubleToInt64Bits(salario) !=
            BitConverter.DoubleToInt64Bits(otro.salario))
        return false;
    return true;
}
```

Reflexividad

No nulificable

Asegurar el tipo para convertir la referencia a su tipo original - **Simetría**

Comparar otros objetos usando su propia definición de equals()

Convertir a Bits la comparación para evitar errores

Transitividad

Consistencia

VB

```
Public Overrides Function Equals(obj As Object) As Boolean
    If Me Is obj Then                                          ← Reflexividad
        Return True
    End If
    If obj Is Nothing Then                                     ← No nulificable
        Return False
    End If
    If Me.GetType() <> obj.GetType() Then                      } Asegurar el tipo para convertir la referencia a su tipo original - Simetría
        Return False
    End If
    Dim otro As Empleado = CType(obj, Empleado)
    If fechaNacimiento Is Nothing Then                         } Comparar otros objetos usando su propia definición de equals()
        If Not otro.fechaNacimiento Is Nothing Then
            Return False
        End If
    ElseIf Not fechaNacimiento.Equals(otro.fechaNacimiento) Then
        Return False
    End If
    If nombre Is Nothing Then
        If Not otro.nombre Is Nothing Then
            Return False
        End If
    ElseIf Not nombre.Equals(otro.nombre) Then
        Return False
    End If
    If BitConverter.DoubleToInt64Bits(salario) <> _            } Convertir a Bits la comparación para evitar errores
            BitConverter.DoubleToInt64Bits(otro.salario) Then
        Return False
    End If
    Return True
End Function
```

Transitividad

Consistencia

Para probar el ejemplo anterior, se implementa una clase que cree dos objetos del mismo tipo y los compare, tanto en sus referencias como en sus atributos:

Ejemplo

C#

```
namespace objeto
{
    class Program
    {
        static void Main(string[] args)
        {
            Fecha f1 = new Fecha(10, 10, 2012);
            Empleado e1 = new Empleado("Juan", f1);
            Empleado e2 = new Empleado("Juan", f1);

            Console.WriteLine(e1.Equals(e2));
            Console.WriteLine(e1.Equals(new Empleado("Juan")));

            if (e1 == e2)
            {
                Console.WriteLine("e1 es identico a e2");
            }
            else
            {
                Console.WriteLine("e1 no es identico a e2");
            }
```

```
            if (e1.Equals(e2))
            {
                Console.WriteLine("e1 es igual a e2");
            }
            else
            {
                Console.WriteLine("e1 no es igual a e2");
            }
            Console.WriteLine("asignar e2 = e1");
            e2 = e1;
            if (e1 == e2)
            {
                Console.WriteLine("e1 es identico a e2");
            }
            else
            {
                Console.WriteLine("e1 no es identico a e2");
            }
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim f1 As New Fecha(10, 10, 2012)
        Dim e1 As New Empleado("Juan", f1)
        Dim e2 As New Empleado("Juan", f1)

        Console.WriteLine(e1.Equals(e2))
        Console.WriteLine(e1.Equals(New Empleado("Juan")))

        If (e1 Is e2) Then
            Console.WriteLine("e1 es identico a e2")
        Else
            Console.WriteLine("e1 no es identico a e2")
        End If
        If (e1.Equals(e2)) Then
            Console.WriteLine("e1 es igual a e2")
        Else
            Console.WriteLine("e1 no es igual a e2")
        End If
        Console.WriteLine("asignar e2 = e1")
        e2 = e1
        If (e1 Is e2) Then
            Console.WriteLine("e1 es identico a e2")
        Else
            Console.WriteLine("e1 no es identico a e2")
        End If
        Console.ReadKey()
    End Sub

End Module
```

El programa que prueba la comparación de objetos presentado anteriormente tiene condicionada su salida por una serie de bifurcaciones condicionales.

La salida que se obtiene es:

```
true
false
e1 no es identico a e2
e1 es igual a e2
asignar e2 = e1
e1 es identico a e2
```

El Método ToString()

Otra de las ventajas que ofrece el hecho que todas las clases hereden de Object es el acceso a uno de sus métodos de utilidad: ToString. Este retorna un String que identifica al objeto y que será acorde a la implementación realizada de la clase que lo define (en otras palabras, como cada clase lo puede sobrescribir, cada una puede implementar su versión del método). Por ejemplo, muchas clases retornan el contenido de sus atributos. Otras retornan información del objeto o crean mensajes descriptivos.

Como en varios casos retornan contenidos de variables, se utiliza para concatenar Strings ya que existen métodos de utilidad que invocan automáticamente a este método. Notoriamente, el método Console.WriteLine está sobrecargado y entre otras cosas recibe un Object como argumento. En su interior, Console.WriteLine invoca a ToString cuando recibe como argumento la referencia a un objeto.

Como el método esta declarado en Object y se puede escribir para adecuarlo a la clase que se cree, cuando se sobrescribe ToString se oculta la visibilidad del método de Object o de otra supercalse que lo haya rescrito y se invoca al sobrescrito en la subclase.

Ejemplo

C#

```
public override string ToString()
{
	return "Empleado [nombre=" + nombre + ", salario=" + salario
		+ ", fechaNacimiento=" + fechaNacimiento + "]";
}
```

VB

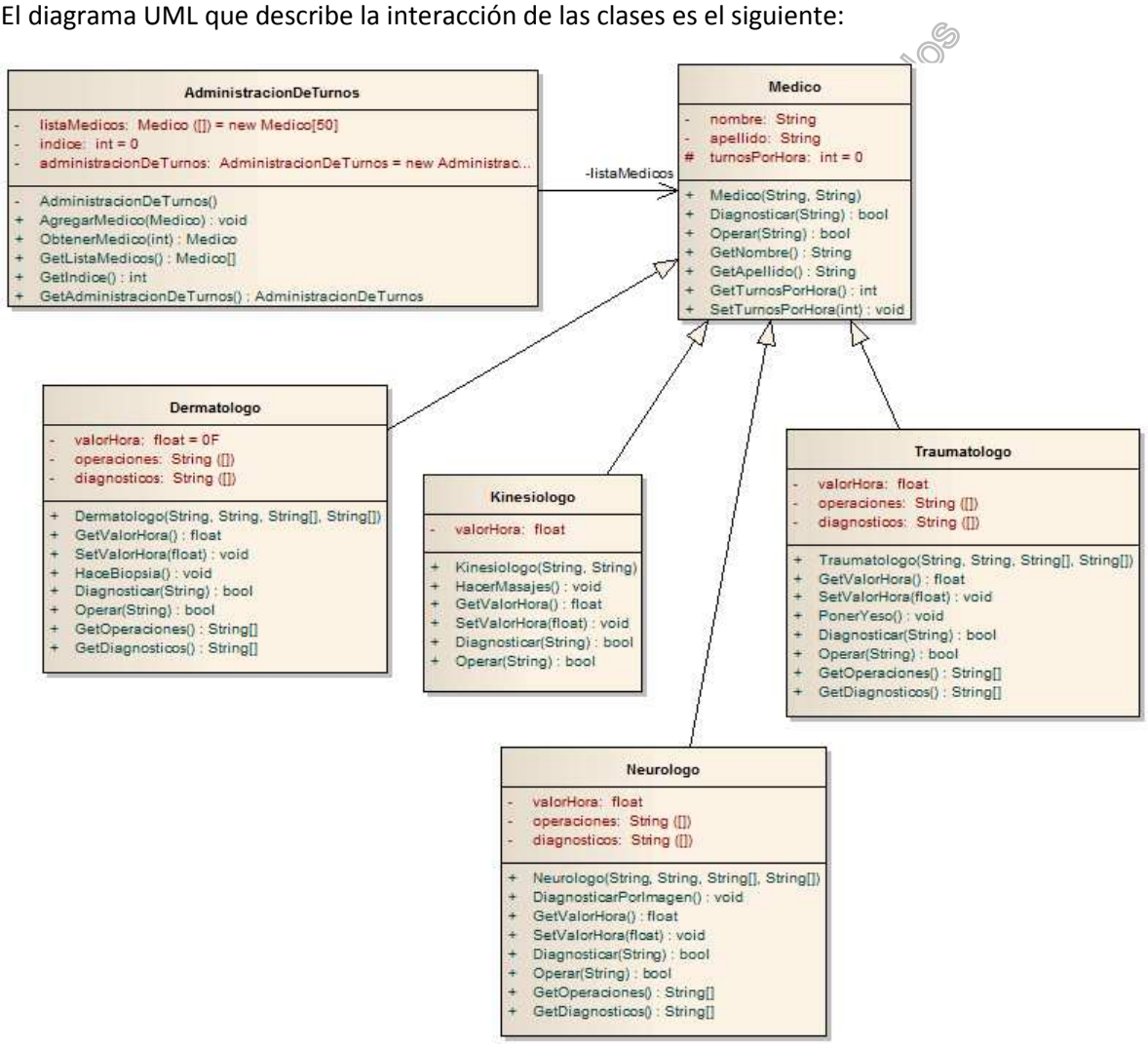
```
Public Overrides Function ToString() As String
	Return "Empleado [nombre=" + nombre + ", salario=" + salario.ToString _
		+ ", fechaNacimiento=" + fechaNacimiento.ToString + "]"
End Function
```

Clases Abstractas

Escenario inicial:

Suponiendo que un sistema necesita generar una clase Medico para modelar los médicos que pertenecen a una clínica. Se sabe que la empresa necesita manejar distintas especialidades en las cuales los médicos prestan servicios como realizar diagnósticos u operaciones (aunque hay algunos que no realizan este servicio).

El diagrama UML que describe la interacción de las clases es el siguiente:



Código inicial

La clase Clinica es la encargada del uso de los objetos del tipo Medico iniciando el valor de su posible creación.

Ejemplo

CS

```
namespace abstractasInicio
{
    class Clinica
    {
        static void Main(string[] args)
        {
            AdministracionDeTurnos adt =
                    AdministracionDeTurnos.getAdministracionDeTurnos();
            String[] operacionest = { "neurofibromatosis", "hernia de disco",
                    "fracturas simples", "fracturas expuestas" };
            String[] diagnosticost = { "lumbalgia", "psoriasis", "fracturas",
                    "dolores", "cervicales" };
            Traumatologo t = new Traumatologo("Iván", "Lacarra", operacionest,
                    diagnosticost);
            adt.agregarMedico(t);

            String[] operacionesd = { "neurofibromatosis", "carcinoma", "lipoma" };
            String[] diagnosticosd = { "liquen plano", "psoriasis", "lepra" };
            Dermatologo d = new Dermatologo("Pedro", "Rivero", operacionesd,
                    diagnosticosd);
            adt.agregarMedico(d);

            Kinesiologo k = new Kinesiologo("Alberto", "Tati");
            adt.agregarMedico(k);

            String[] operacionesn = { "neurofibromatosis", "neurinoma" };
            String[] diagnosticosn = { "vértigo", "psoriasis", "náuseas", "ACV" };
            Neurologo n = new Neurologo("Pedro", "Pomar", operacionesn,
              diagnosticosn);
            adt.agregarMedico(n);

            ReporteDeMedicos rm = new ReporteDeMedicos();
            rm.generarReporteActividades();

            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class Clinica

    Public Sub InicioOperaciones()
        Dim adt As AdministracionDeTurnos = _
             AdministracionDeTurnos.getAdministracionDeTurnos()

        Dim operacionest() As String = {"neurofibromatosis", "hernia de disco", _
              "fracturas simples", "fracturas expuestas"}
        Dim diagnosticost() As String = {"lumbalgia", "psoriasis", "fracturas", _
              "dolores", "cervicales"}
        Dim t As New Traumatologo("Iván", "Lacarra", operacionest, diagnosticost)
```

```
        adt.agregarMedico(t)

        Dim operacionesd() As String = {"neurofibromatosis", "carcinoma", "lipoma"}
        Dim diagnosticosd() As String = {"liquen plano", "psoriasis", "lepra"}
        Dim d As New Dermatologo("Pedro", "Rivero", operacionesd, diagnosticosd)
        adt.agregarMedico(d)

        Dim k As New Kinesiologo("Alberto", "Tati")
        adt.agregarMedico(k)

        Dim operacionesn() As String = {"neurofibromatosis", "neurinoma"}
        Dim diagnosticosn() As String = {"vértigo", "psoriasis", "náuseas", "ACV"}
        Dim n As New Neurologo("Pedro", "Pomar", operacionesn, diagnosticosn)
        adt.agregarMedico(n)

        Dim rm As New ReporteDeMedicos()
        rm.generarReporteActividades()

        Console.ReadKey()
    End Sub

End Class
```

Código de ReporteDeMedicos

La clase encargada de generar los reportes necesita acceder a la clase de instancia única que tiene a su cargo la administración de turnos y doctores de la clínica.

Ejemplo

C#

```
namespace abstractasInicio
{
    class ReporteDeMedicos
    {
        public void generarReporteActividades()
        {
            AdministracionDeTurnos adt = AdministracionDeTurnos
                    .getAdministracionDeTurnos();
            Medico[] listaMedicos = adt.getListaMedicos();
            int indice = adt.getIndice();
            String tipoDiagnostico = "psoriasis";
            String tipoOperacion = "neurofibromatosis";

            for (int i = 0; i < indice; i++)
            {
                Console.WriteLine("El médico " + listaMedicos[i].getNombre() + " "
                        + listaMedicos[i].getApellido());
                if (listaMedicos[i].operar(tipoOperacion))
                    Console.WriteLine("Opera " + tipoOperacion);
                else
                    Console.WriteLine("No opera");
                if (listaMedicos[i].diagnosticar(tipoDiagnostico))
                    Console.WriteLine("Diagnostica " + tipoDiagnostico);
```

```
                else
                    Console.WriteLine("No diagnostica");
            }
        }
    }
}
```

VB

```
Public Class ReporteDeMedicos
    Public Sub generarReporteActividades()
        Dim adt As AdministracionDeTurnos = _
             AdministracionDeTurnos.getAdministracionDeTurnos()
        Dim listaMedicos() As Medico = adt.getListaMedicos()
        Dim indice As Integer = adt.getIndice()
        Dim tipoDiagnostico As String = "psoriasis"
        Dim tipoOperacion As String = "neurofibromatosis"

        For i As Integer = 0 To indice - 1
            Console.WriteLine("El médico " + listaMedicos(i).Nombre + " " _
                + listaMedicos(i).Apellido)
            If (listaMedicos(i).operar(tipoOperacion)) Then
                Console.WriteLine("Opera " + tipoOperacion)
            Else
                Console.WriteLine("No opera")
            End If
            If (listaMedicos(i).diagnosticar(tipoDiagnostico)) Then
                Console.WriteLine("Diagnostica " + tipoDiagnostico)
            Else
                Console.WriteLine("No diagnostica")
            End If
        Next
    End Sub
End Class
```


El problema de este código radica en que sólo las subclases tendrán “el conocimiento” de cómo calcular tanto los diagnósticos como las operaciones que realizan los médicos. Con las técnicas explicadas hasta el momento, se debe crear un método con cuerpo vacío para que las subclases lo puedan rescribir.

Definiciones

Una clase abstracta modela un tipo de objetos donde algunos métodos deben ser especificados en la subclase porque su funcionamiento es inherente a ellas.

Algunas veces, una clase que se ha definido representa un concepto abstracto y como tal, no debe ser implementado en ese momento.

En la programación orientada a objetos, se podría modelar conceptos abstractos pero no querer que se creen implementaciones de ellos. Existen clases en .Net que permiten crear modelos en este tipo de situaciones, las cuales definen conceptos abstractos y no deben ser implementadas

(no se deben crear objetos de su tipo) que son llamadas clases abstractas. Una clase abstracta es una clase que sólo puede tener subclases, no puede ser instanciada.

Las clases abstractas se definen mediante el modificador `abstract` o `MustInherit` (C# o VB).

Las clases abstractas están estrechamente relacionadas con las interfaces. Una diferencia clave entre las clases abstractas e interfaces es que una clase puede implementar un número ilimitado de interfaces, pero sólo puede heredar sólo de una clase abstracta (o de cualquier otro tipo). Una clase que se deriva de una clase abstracta también puede implementar sus propias interfaces. Las clases abstractas son útiles cuando se crean componentes ya que le permiten especificar un nivel invariable de funcionalidad en algunos métodos, pero dejan al código que las utilicen otros métodos que una implementación específica de esa clase necesita. También son muy buenas para crear versiones, porque si se necesita funcionalidad adicional en las clases derivadas, se puede añadir a la clase base sin romper el código.

Los métodos que están destinados a ser invariantes puede ser codificados en la clase base, pero cualquiera de los métodos que se han de de implementar en las subclases están marcados con los modificadores `abstract` o `MustOverride` (C# o VB).

Si se intenta crear un objeto del tipo de la clase abstracta, el compilador mostrará un error y no compilará el programa.

Las clases abstractas tienen las siguientes características:

- Una clase abstracta no puede ser instanciada.
- Una clase abstracta puede contener métodos abstractos y descriptores de acceso.
- No es posible modificar una clase abstracta con el modificador `sealed` u `NotOverridable` (C# o VB), lo que significa que la clase no se puede heredar.
- Una clase no abstracta derivada de otra clase abstracta debe incluir implementaciones reales de todos los métodos abstractos heredados y descriptores de acceso.
- El modificador `abstract` o `MustOverride` (C# o VB) se puede utilizar con las clases, métodos, propiedades, indizadores y eventos.
- Una clase abstracta que implementa una interfaz puede asignar los métodos de interfaz a métodos abstractos. Es decir, puede redefinir los métodos de la interfaz como abstractos dentro de la clase abstracta.

Métodos y propiedades abstractos

Una clase abstracta debe contener propiedades o métodos abstractos, esto es, que no tienen implementación. De esta forma, una clase abstracta puede definir un interfaz de programación completa, incluso proporciona a sus subclases la declaración de todos los métodos necesarios para implementar la interfaz de programación. Sin embargo, las clases abstractas pueden dejar algunos detalles o toda la implementación de aquellos métodos a sus subclases.

Una clase abstracta en la cual sólo se definen métodos abstractos se denomina clase abstracta pura

Los métodos abstractos tienen las siguientes características:

- Un método abstracto es implícitamente un método virtual.
- Las declaraciones de métodos abstractos sólo se permiten en las clases abstractas.
- Como una declaración de método abstracto no proporciona una implementación, no hay cuerpo del método (bloque de sentencias asociado), la declaración del método simplemente termina con un punto y coma y no hay llaves ({}) después de la firma.
- La implementación la proporciona un método de que rescribirá al actual ocultándolo, el cual es un miembro de una clase no abstracta.
- Es un error utilizar los modificadores `static` o `virtual` en la declaración de un método abstracto.

Las propiedades abstractas se comportan como métodos abstractos, a excepción de las diferencias en la declaración y la sintaxis de invocación:

- Es un error utilizar el modificador `abstract` en una propiedad estática.
- Una propiedad abstracta heredada se puede reemplazar en una clase derivada mediante la inclusión de una declaración de propiedad que utiliza el modificador `override`.

Para mostrar un ejemplo de cuando sería necesario crear una clase abstracta con métodos abstractos se puede considerar que en una aplicación de dibujo orientada a objetos, se pueden dibujar círculos, rectángulos, líneas, etc... Cada uno de esos objetos gráficos comparten ciertos estados (posición, caja de dibujo) y comportamiento (movimiento, redimensionado, dibujo). Se pueden aprovechar esas similitudes y declararlos todos a partir de un mismo objeto padre, como ser ObjetoGrafico.

Sin embargo, los objetos gráficos también tienen diferencias substanciales: dibujar un círculo es bastante diferente a dibujar un rectángulo. Los objetos gráficos no pueden compartir estos tipos de estados o comportamientos. Por otro lado, todos los ObjetosGraficos deben saber como dibujarse a sí mismos; se diferencian en cómo se dibujan unos y otros. Esta es la situación perfecta para una clase abstracta.

Primero se debe declarar una clase abstracta, ObjetoGrafico, para proporcionar las variables miembro y los métodos que van a ser compartidos por todas las subclases, como la posición actual y el método MoverA().

También se deberían declarar métodos abstractos como Dibujar(), que necesita ser implementado por todas las subclases, pero de manera completamente diferente (no tiene sentido crear una implementación por defecto en la superclase). La clase ObjetoGrafico se con sus declaraciones más elementales se miestra a continuación.

Ejemplo

C#

```
namespace graficos
{
    public abstract class ObjetoGrafico
    {
        int x, y;

        public void MoverA(int nuevaX, int nuevaY)
        {
            Console.WriteLine("Moviendo a (" + nuevaX + ", " + nuevaY + ")");
        }
        public abstract void Dibujar();
    }
}
```

VB

```
Public MustInherit Class ObjetoGrafico
    Private _x As Integer
    Private _y As Integer

    Public Sub MoverA(nuevaX As Integer, nuevaY As Integer)
        Console.WriteLine("Moviendo a (" + nuevaX + ", " + nuevaY + ")")
    End Sub

    Public MustOverride Sub Dibujar()
End Class
```

Todas las subclases no abstractas de ObjetoGrafico como son Circulo o Rectangulo deberán proporcionar una implementación para el método Dibujar(). El código a continuación muestra esta situación.

Ejemplo

C#

```
namespace graficos
{
    class Circulo : ObjetoGrafico
    {
        public override void Dibujar()
        {
            Console.WriteLine("Dibujando el círculo");
        }
    }
}

namespace graficos
{
    class Rectangulo : ObjetoGrafico
    {
        public override void Dibujar()
        {
```

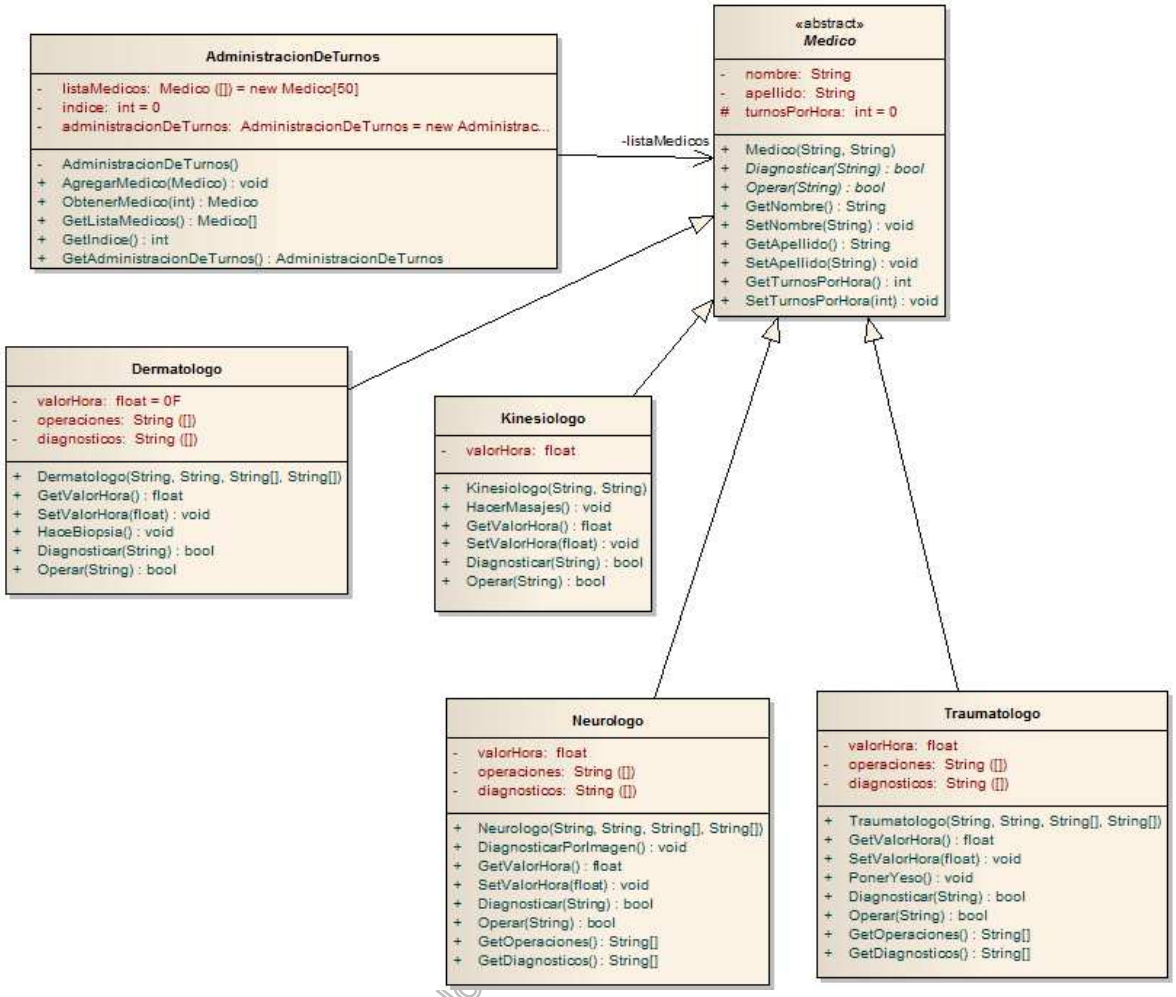
```
            Console.WriteLine("Dibujando el rectángulo");
        }
    }
}
```

VB

```
Public Class Circulo
    Inherits ObjetoGrafico

    Public Overrides Sub Dibujar()
        Console.WriteLine("Dibujando el círculo")
    End Sub
End Class

Public Class Rectangulo
    Inherits ObjetoGrafico

    Public Overrides Sub Dibujar()
        Console.WriteLine("Dibujando el rectángulo")
    End Sub
End Class
```

Una subclase que hereda de una superclase se sigue considerando abstracta hasta que todos los métodos abstractos en ella dejen de serlo.

Un ejemplo de solución al problema anterior de los médicos se muestra en el gráfico



Ejemplo

C#

```
namespace abstractasFin
{
    public abstract class Medico
    {

        private String nombre;
        private String apellido;
        protected int turnosPorHora = 0;

        public Medico(String nombre, String apellido)
        {
            this.nombre = nombre;
            this.apellido = apellido;
        }

        public abstract bool Diagnosticar(String tipoDiagnostico);
```

```
        public abstract bool Operar(String tipoOperacion);

        public String GetNombre()
        {
            return nombre;
        }

        public void SetNombre(String nombre)
        {
            this.nombre = nombre;
        }

        public String GetApellido()
        {
            return apellido;
        }

        public void SetApellido(String apellido)
        {
            this.apellido = apellido;
        }

        public int GetTurnosPorHora()
        {
            return turnosPorHora;
        }

        public void SetTurnosPorHora(int turnosPorHora)
        {
            this.turnosPorHora = turnosPorHora;
        }
    }
}
```

VB

```
Public MustInherit Class Medico
    Private _nombre As String
    Private _apellido As String
    Private _turnosPorHora As Integer = 0

    Public Sub New(nombre As String, apellido As String)
        Me._nombre = nombre
        Me._apellido = apellido
    End Sub

    Public MustOverride Function Diagnosticar(tipoDiagnostico As String) As Boolean

    Public MustOverride Function Operar(tipoOperacion As String) As Boolean

    Public ReadOnly Property Nombre() As String
        Get
            Return _nombre
        End Get
    End Property
```

```
    Public ReadOnly Property Apellido() As String
        Get
            Return _apellido
        End Get
    End Property

    Public Property TurnosPorHora() As Integer
        Get
            Return _turnosPorHora
        End Get
        Set(ByVal value As Integer)
            _turnosPorHora = value
        End Set
    End Property
End Class
```

Cada de uno de los médicos posibles se debería implementar de forma similar a la siguiente clase de ejemplo.

Ejemplo

C#

```
namespace abstractasFin
{
    class Dermatologo : Medico
    {
        private float valorHora = 0F;
        private String[] operaciones;
        private String[] diagnosticos;

        public Dermatologo(String nombre, String apellido, String[] operaciones,
                String[] diagnosticos)
            : base(nombre, apellido)
        {
            this.operaciones = operaciones;
            this.diagnosticos = diagnosticos;
        }

        public float GetValorHora()
        {
            return valorHora;
        }

        public void SetValorHora(float valorHora)
        {
            this.valorHora = valorHora;
        }

        public void HaceBiopsia()
        {

        }

        public override bool Diagnosticar(String tipoDiagnostico)
```

```
        {
            for (int i = 0; i < diagnosticos.Length; i++)
            {
                if (diagnosticos[i] != null
                        && diagnosticos[i].Equals(tipoDiagnostico))
                    return true;
            }
            return false;
        }

        public override bool Operar(String tipoOperacion)
        {
            for (int i = 0; i < operaciones.Length; i++)
            {
                if (operaciones[i] != null && operaciones[i].Equals(tipoOperacion))
                    return true;
            }
            return false;
        }
    }
}
```

VB

```
Public Class Dermatologo
    Inherits Medico
    Private _valorHora As Single = 0.0F
    Private _operaciones() As String
    Private _diagnosticos() As String

    Public Sub New(_nombre As String, _apellido As String, _operaciones() _
             As String, _diagnosticos() As String)
        MyBase.New(_nombre, _apellido)
        Me._operaciones = _operaciones
        Me._diagnosticos = _diagnosticos
    End Sub

    Public Overrides Function Diagnosticar(tipoDiagnostico As String) As Boolean
        For i As Integer = 0 To _diagnosticos.Length - 1
            If (Not _diagnosticos(i) Is Nothing) And _
                    _diagnosticos(i).Equals(tipoDiagnostico) Then
                Return True
            End If
        Next
        Return False
    End Function

    Public Overrides Function Operar(tipoOperacion As String) As Boolean
        For i As Integer = 0 To _operaciones.Length - 1
            If (Not _operaciones(i) Is Nothing) And _
                    _operaciones(i).Equals(_operaciones) Then
                Return True
            End If
        Next
        Return False
    End Function
```

```
    Public ReadOnly Property Operaciones() As String()
        Get
            Return _operaciones
        End Get
    End Property
    Public Property ValorHora() As Single
        Get
            Return _valorHora
        End Get
        Set(ByVal value As Single)
            _valorHora = value
        End Set
    End Property
    Public ReadOnly Property Diagnosticos() As String()
        Get
            Return _diagnosticos
        End Get
    End Property

    Public Sub HaceBiopsia()

    End Sub
End Class
```

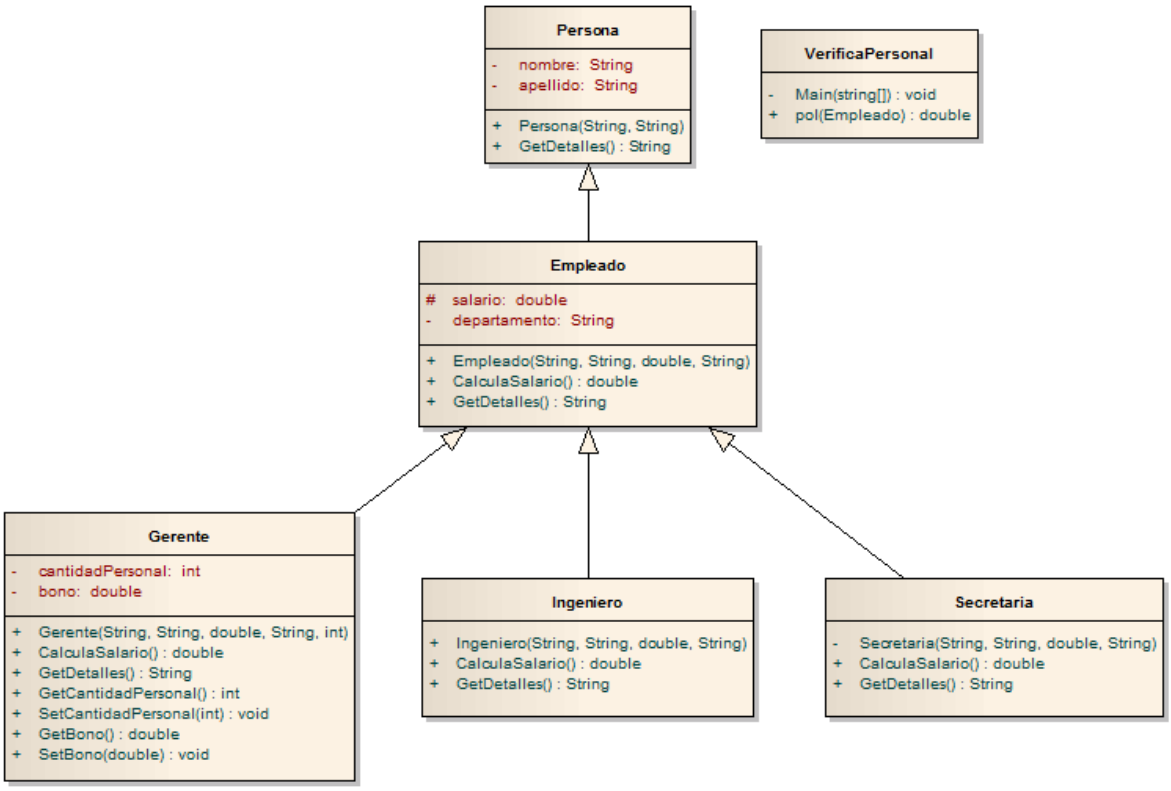
Polimorfismo

El polimorfismo se manifiesta cuando se conoce la operación a realizar pero se desconoce cuando se diseña un programa que objeto la va a llevar a cabo. En .Net esto se produce cuando se invoca a un método conocido luego de asignar la referencia al objeto que se desea ejecute la acción.

La forma más sencilla de ver esto es a través de un ejemplo como el que se muestra en el siguiente diagrama UML que plantea una serie de herencias (tres para ser más exactos) y métodos en común en las distintas clases que las componen.

En el ejemplo existen dos métodos que se sobrescriben, uno, GetDetalles, desde la superclase base de todas las cadenas de herencias, Persona, y otro desde la clase Empleado, CalculaSalario. En base a la reescritura de estos métodos y el ocultamiento de la visibilidad que se produce por ella, se plantearán los conceptos base del polimorfismo.

Ejemplo



Cuando una acción (método) depende del objeto que la ejecuta, se lo denomina polimorfismo, porque tiene muchas "formas" de ejecutarse. Cada forma de ejecutarse estará determinada por el comportamiento del objeto en el cual fue implementado el método.

Los objetos tienen una sola forma, pero las acciones que se ejecuten en ellos pueden ser específicas de cada uno. De esta manera la acción GetDetalles() se llamará igual en las clases Persona, Empleado y Gerente, pero el resultado como la acción en si misma será diferente. Lo mismo ocurre en el caso del método CalculaSalario. Notar que este último se empieza a implementar a partir de la clase Empleado, por lo tanto, un método no debe estar en la superclase de la cadena de herencia para poder implementar polimorfismo

Una referencia puede serlo de distintos objetos, pero si la acción en todos ellos es la misma, cambiando la referencia se indica sobre que objeto se ejecutará la acción, por lo tanto, el hecho de conocer la acción permite valerse de una variable a la que se le asignará la referencia al objeto en particular que se quiera invocar, para luego utilizar con notación de punto la invocación al mismo.

Invocación Virtual de Métodos

Para realizar una invocación virtual de métodos (virtual porque en tiempo de diseño no se conoce a que objeto pertenece el método a ejecutar) sólo se necesita asignar la referencia al objeto que

tiene en método a ejecutar. Esto si bien brinda un gran poder de ejecución (la referencia “virtual” se resuelve en tiempo de ejecución en el momento de asignarle valor a la variable que se utilizará para realizar la invocación del método) tiene ciertas limitaciones que deben ser tenidas en cuenta al momento de programar.

Por ejemplo, cuando se declara una referencia, se debe recordar que esta permitido asignar valores de subclases a referencias de superclases, pero no a la inversa (al menos, no sin hacer una conversión de tipo explícita). Por lo tanto, si este es el caso, sólo se podrá invocar a los métodos que “ve” la variable de referencia. En otras palabras, sólo se podrán acceder a métodos que estén declarados en la superclase y rescritos en las subclases.

Cualquier intento de acceder a un elemento de la subclase que no este definido en la superclase será un error en tiempo de compilación.

El polimorfismo es referido a menudo como el tercer pilar de la programación orientada a objetos, después de la encapsulación y herencia. Es una palabra griega que significa "de muchas en formas" y tiene dos aspectos distintos:

- En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como si fueran del tipo de su clase base en lugares como: parámetros de métodos, colecciones o vectores. Cuando esto ocurre, el tipo del objeto en tiempo de ejecución ya no es idéntico a su tipo declarado (esto sucede al asignar referencias de la subclase a referencias de la superclase, lo cual deriva en una conversión de tipo).
- Las clases base pueden definir y aplicar métodos virtuales (`virtual` o `MustOverride` – C# o VB-) y las clases derivadas pueden rescribirlos, lo que significa que proporcionan su propia definición e implementación. En tiempo de ejecución, cuando el código cliente llama al método, CLR busca el tipo en tiempo de ejecución del objeto, e invoca la rescritura del método virtual. Así, en el código fuente se puede llamar a un método en una clase base, y causar la ejecución de la versión del método en una clase derivada.

Los métodos virtuales permiten trabajar con grupos de objetos relacionados de una manera uniforme. Por ejemplo, suponer que se tiene una aplicación de dibujo que permite a un usuario crear varios tipos de formas en una superficie para dibujar. No se sabe en tiempo de compilación que tipos de formas determinadas el usuario va a crear. Sin embargo, la aplicación tiene que llevar un registro de todos los diferentes tipos de formas que se crean y tiene que actualizarlos en respuesta a las acciones del usuario. Se puede utilizar polimorfismo de resolver este problema de dos pasos básicos:

- Crear una serie de cadenas de herencia de clases en la que cada clase de forma específica se deriva de una clase base común.

- Usar un método `virtual` / `abstract` o `Overridable` / `MustOverride` (C# o VB) para invocar el método apropiado en cualquier clase derivada a través de una simple llamada al método de la clase base.

Retomando el ejemplo usado para definir clases, métodos y propiedades abstractas, se puede ampliar para ver como trabajar polimórficamente los métodos en las clases derivadas a través de llamadas y conversiones de tipo. Recordar que un método abstracto es implícitamente virtual, por lo tanto se puede redefinir por medio de la rescritura en las clases derivadas.

Ejemplo

C#

```
namespace graficos
{
    class Program
    {
        static void Main(string[] args)
        {
            Rectangulo r = new Rectangulo();
            Circulo c = new Circulo();
            MetodoPolimorfico(r);
            MetodoPolimorfico(c);
            Console.ReadKey();
        }

        public static void MetodoPolimorfico(ObjetoGrafico o)
        {
            o.Dibujar();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim r As New Rectangulo()
        Dim c As New Circulo()
        MetodoPolimorfico(r)
        MetodoPolimorfico(c)
        Console.ReadKey()
    End Sub
    Public Sub MetodoPolimorfico(o As ObjetoGrafico)
        o.Dibujar()
    End Sub

End Module
```


Estas situaciones son comunes en cualquier tipo de escenario, no sólo en un ambiente gráfico. Por ejemplo, si se desea aplicar polimorfismo para obtener acceso a acciones en cadenas de herencia que representen empleados, se los puede invocar de manera análoga. Recordar que el

polimorfismo se basa en conversiones de tipo y cuando una referencia se convierte al tipo de la superclase, esta puede acceder a los elementos definidos en la clase base que tengan un modificador de acceso que lo permita, pero no podrá acceder a los elementos de la subclase más allá del modificador que esta posea, porque simplemente las referencias del tipo de la clase base no acceden nunca a los elementos de las subclases.

Cuando las asignaciones están correctas, se pueden hacer invocaciones virtuales (referencias polimórficas). A esto se lo denomina "**polimorfismo por herencia**".

Ejemplo

C#

```
Empleado e = new Gerente("Pedro", "Gonzalez", 10000, "Sistemas", 15); //legal
                :
                :
e.GetDetalles();
                :
                :
```

VB

```
Dim e As Empleado = New Gerente("Pedro", "Gonzalez", 10000, "Sistemas", 15)'legal
                :
                :
e.GetDetalles();
                :
                :
```

Para valerse de la oportunidad que brinda el ocultar un método cuando está rescrito, se deben crear objetos de las clases que se utilizarán. Por ejemplo, dentro de la clase empleado se puede ver uno de los métodos rescritos.

Ejemplo

C#

```
namespace polimorfismo
{
    class Empleado : Persona
    {
                :
                :

        public virtual double CalculaSalario()
        {
            return salario = salario - salario * 0.21;
        }
                :
                :
    }
}
```

VB

```
Public Class Empleado
    Inherits Persona
              :
              :

    Public Overridable Function CalculaSalario() As Double
        _salario = _salario - _salario * 0.20999999999999999
        Return _salario
    End Function
              :
              :
End Class
```

También se puede observar el mismo método dentro de la clase Gerente

C#

```
namespace polimorfismo
{
    class Gerente : Empleado
    {
              :
              :

        public override double CalculaSalario()
        {
            return salario = salario - salario * 0.21 + bono;
        }

              :
              :
    }
}
```

VB

```
Public Class Gerente
    Inherits Empleado
              :
              :

    Public Overrides Function CalculaSalario() As Double
        _salario = _salario - _salario * 0.20999999999999999 + bono
        Return _salario
    End Function
              :
              :
End Class
```

Para utilizar la capacidad de invocar métodos virtuales, una tercera clase crea objetos de Empleado y Gerente para valerse del ocultamiento de métodos por rescritura al momento de realizar la invocación. Para ello utiliza la superclase de la sub cadena de herencia a partir de la

clase Empleado y declara una referencia de este tipo para asignarle valores de referencias de objetos de su tipo o de los de una subclase de ella.

Ejemplo

Ejemplo

C#

```
namespace polimorfismo
{
    class VerificaPersonal
    {
        static void Main(string[] args)
        {
            VerificaPersonal vp = new VerificaPersonal();
            // Empleado e = new Gerente("Pedro", "Gonzalez", 10000, "Sistemas", 15);
            // //legal
            // e.SetCantidadPersonal(15); // Intento ilegal de acceder a
            // un atributo del tipo Gerente
            // La variable de referencia es declarada como un tipo
            // Empleado, por lo tanto puede acceder a elementos que
            // existan en Empleado, aunque el objeto sea de tipo Gerente

            // e.GetDetalles();
            Gerente g = new Gerente("Juan", "Perez", 10000, "Sistemas", 15);
            Empleado e = new Empleado("Sebastián", "Dominguez", 5000, "Sistemas");
            double t = vp.Pol(g);
            Console.WriteLine("Empleado: " + g.GetDetalles());
            Console.WriteLine("Tiene un sueldo de: " + t);
            t = vp.Pol(e);
            Console.WriteLine("Empleado: " + e.GetDetalles());
            Console.WriteLine("Tiene un sueldo de: " + t);
            Console.ReadKey();
        }

        public double Pol(Empleado e)
        {
            return e.CalculaSalario();
        }
    }
}
```

VB

```
Public Class VerificaPersonal
    Public Sub InicioOperaciones()
       ' Dim e As Empleado = New Gerente("Pedro", "Gonzalez", 10000, "Sistemas", 15)
        ' 'legal
        ' e.SetCantidadPersonal(15) ' Intento ilegal de acceder a
        ' un atributo del tipo Gerente
        ' La variable de referencia es declarada como un tipo
        ' Empleado, por lo tanto puede acceder a elementos que
        ' existan en Empleado, aunque el objeto sea de tipo Gerente
```

```
        ' e.GetDetalles()
        Dim g As New Gerente("Juan", "Perez", 10000, "Sistemas", 15)
        Dim e = New Empleado("Sebastián", "Dominguez", 5000, "Sistemas")
        Dim t As Double = Pol(g)
        Console.WriteLine("Empleado: " + g.GetDetalles())
        Console.WriteLine("Tiene un sueldo de: " + t.ToString)
        t = Pol(e)
        Console.WriteLine("Empleado: " + e.GetDetalles())
        Console.WriteLine("Tiene un sueldo de: " + t.ToString)
        Console.ReadKey()
    End Sub

    Public Function Pol(e As Empleado) As Double
        Return e.CalculaSalario()
    End Function
End Class
```

Interfaces

Las interfaces, como las clases, definen un conjunto de propiedades, métodos y eventos. Pero a diferencia de las clases, las interfaces no proporcionan la implementación. Se implementan en las clases, y se definen como entidades separadas de las clases.

Una interfaz representa un contrato, en que una clase que implementa una interfaz debe hacer concretos (escribir el código asociado a cada elemento) todos los aspectos de esa interfaz tal y como está definida.

Con interfaces, se pueden definir características comunes como pequeños grupos de miembros estrechamente relacionados. Se pueden desarrollar implementaciones mejoradas para las interfaces sin poner en peligro el código existente, reduciendo así al mínimo los problemas de compatibilidad. También se puede agregar nuevas características en cualquier momento mediante el desarrollo de interfaces adicionales e implementaciones.

Aunque las implementaciones de las interfaces pueden evolucionar, las interfaces en sí mismas no se deberían cambiar una vez publicadas. Los cambios en éstas pueden romper el código existente. Si se piensa en una interfaz como un contrato, es evidente que ambas partes del contrato tienen un papel que desempeñar. El creador de una interfaz está de acuerdo en no volver a cambiar esa interfaz, y el implementador se compromete a crear el código correspondiente a la interfaz exactamente como fue diseñada.

En las versiones anteriores de Visual Basic, se puede utilizar interfaces pero no crearlas directamente. Ahora se puede definir interfaces reales utilizando la instrucción `Interface`, y se puede implementar interfaces con una versión mejorada de la palabra clave `Implements`.

Una interfaz pública (en realidad este es su único posible modificador ya que no existen interfaces de otra clase de visibilidad) es un convenio con la clase que la implementa, la cual deberá escribir el código que se ejecutará en el método cuyo prototipo se declara en la interfaz.

Este convenio es el que permite, a través de una referencia en común entre dos clases que la implementan, realizar llamados virtuales a métodos, propiedades o eventos. La referencia en común es justamente del tipo de la interfaz (se pueden crear variables de referencias del tipo de una interfaz, lo que no se puede hacer es crear objetos de su tipo porque los métodos no tienen bloques de sentencias asociados, es decir, son abstractos a pesar de no existir una declaración explícita en el código de este hecho).

Por lo tanto, una interfaz es la declaración formal del contrato, de manera que en ella sólo se encuentran los prototipos de los métodos a implementar en la clase y por ser estos "abstractos". La clase que implemente la interfaz tiene obligación de rescribirlos o no se podrán crear objetos de su tipo (debe cumplir con el "contrato").

Muchas clases no relacionadas pueden implementar una interfaz y de esta manera utilizarla como "puente de referencias" para llamar las acciones con el mismo nombre en clases que no estén en la misma cadena de herencia pero que implementen la misma interfaz.

Una clase puede implementar muchas interfaces, la única salvedad es que para realizar esto cada declaración deberá separarse con comas.

Declaran métodos que se esperan que una o más clases los implementen, o, en otras palabras, que las clases que implementen la interfaz provean el cuerpo del método o bloque de sentencias asociado.

Determinan la interfaz del objeto cuya clase la implemente sin revelar el cuerpo de la clase misma. Con sólo conocer los métodos que existen en la interfaz y las clases que la implementan, es suficiente para realizar invocaciones de los mismos en objetos del tipo de dichas clases y no es necesario brindar ninguna otra información de los servicios que prestan los objetos del tipo de las clases que implementan la interfaz, mucho menos aún de su estructura interna.

Capturan similitudes entre clases no relacionadas sin forzar relaciones de asociación, agregación, composición o herencia entre ellas, o, mejor dicho, cuando dos clases realizan las mismas acciones pero pertenecen a cadenas de herencia diferente, pueden compartir invocaciones virtuales de métodos si implementan la misma interfaz y luego valerse de declarar una variable del tipo de la interfaz y asignarle las referencias a los objetos de los que se quiera utilizar sus servicios.

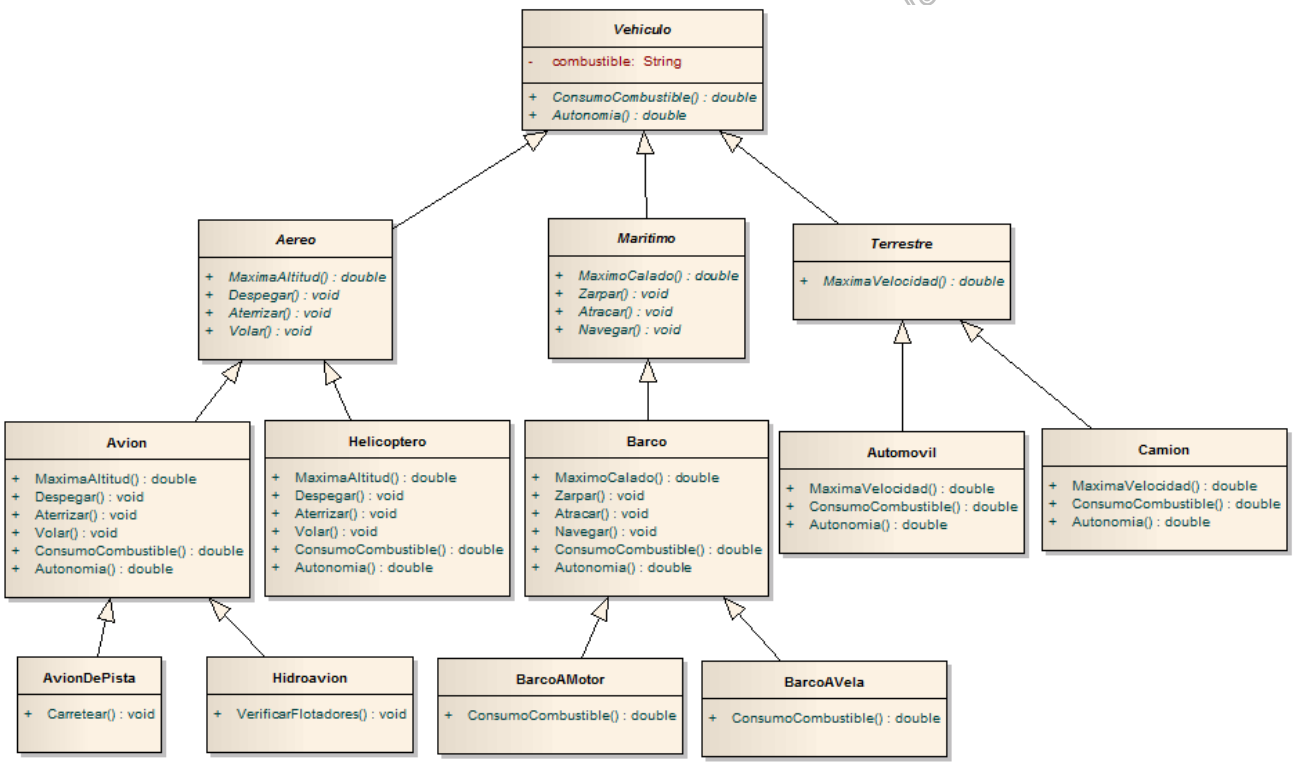
Simula la herencia múltiple porque se pueden implementar muchas interfaces. Si se condiciona a que sólo se puede hacer herencia múltiple si se deriva de una sola clase concreta y tantas clases abstractas puras (clases que no poseen ningún método concreto) como se desee, se estaría frente al caso de la herencia simple e interfaces.

Estas invocaciones son polimórficas respecto de la referencia del tipo de la interfaz y al utilizar las invocaciones de los métodos que implementan las mismas se lo conoce como "**polimorfismo por interfaces**"

Ejemplo

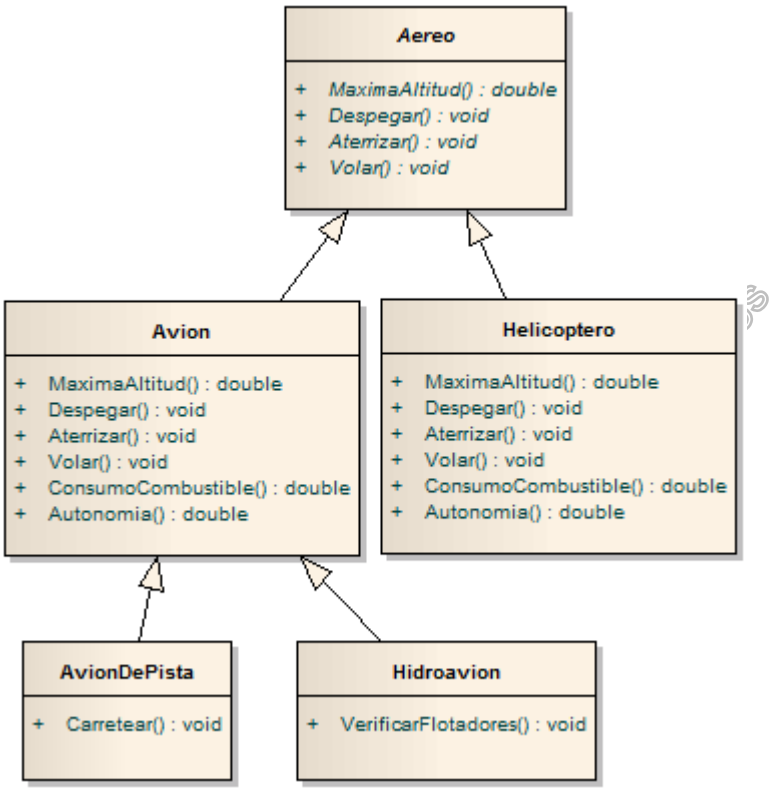
De la misma forma utilizada para entender el polimorfismo, es más fácil comprender las posibilidades brindadas por las interfaces a través de un ejemplo. Como punto de partida se supone un diseño de herencias de clases como muestra el gráfico UML. Recordar que este tipo de gráficos no necesita exponer todos los miembros de clases, sólo los más importantes.

Ejemplo



Dentro del contexto del ejemplo, sólo interesa analizar una cadena de herencia, en la cuál se puede observar los métodos que se implementan en la clase Aéreo. Si se mira bien esta clase, se puede apreciar que posee algunos métodos abstractos y que la clase en si misma también lo es. Esto es lógico si tenemos en cuenta que la información acerca de cómo realizar estas acciones estará disponible en las subclases.

Ejemplo



Refinamiento

Cuando se analizan con detalles este tipo de clases se puede detectar métodos que reflejan acciones que se pueden llevar a cabo por más de un tipo de objeto y, además, muchos de ellos ni siquiera están en la misma cadena de herencia. Este hecho se puede detectar en la etapa de análisis, de diseño o, inclusive, en una etapa de desarrollo cuando se compara con otras cadenas de herencia.

Cuando las clases definen objetos de distintos tipo que realizan las mismas acciones se descubren comportamientos similares entre objetos. Estos se pueden relacionar a través de una referencia en común donde se declaren dichas acciones.

Con esta referencia, se puede llamar a esos métodos cuando los objetos (del mismo tipo u otro en donde están declarados los métodos) deben brindar un servicio que necesita dichas operaciones para llevarse a cabo.

Para crear esa referencia en común, se extraen en un refinamiento los métodos que tengan estas características y se los coloca en una interfaz, para que luego cada clase la implemente y rescriba dichos métodos.

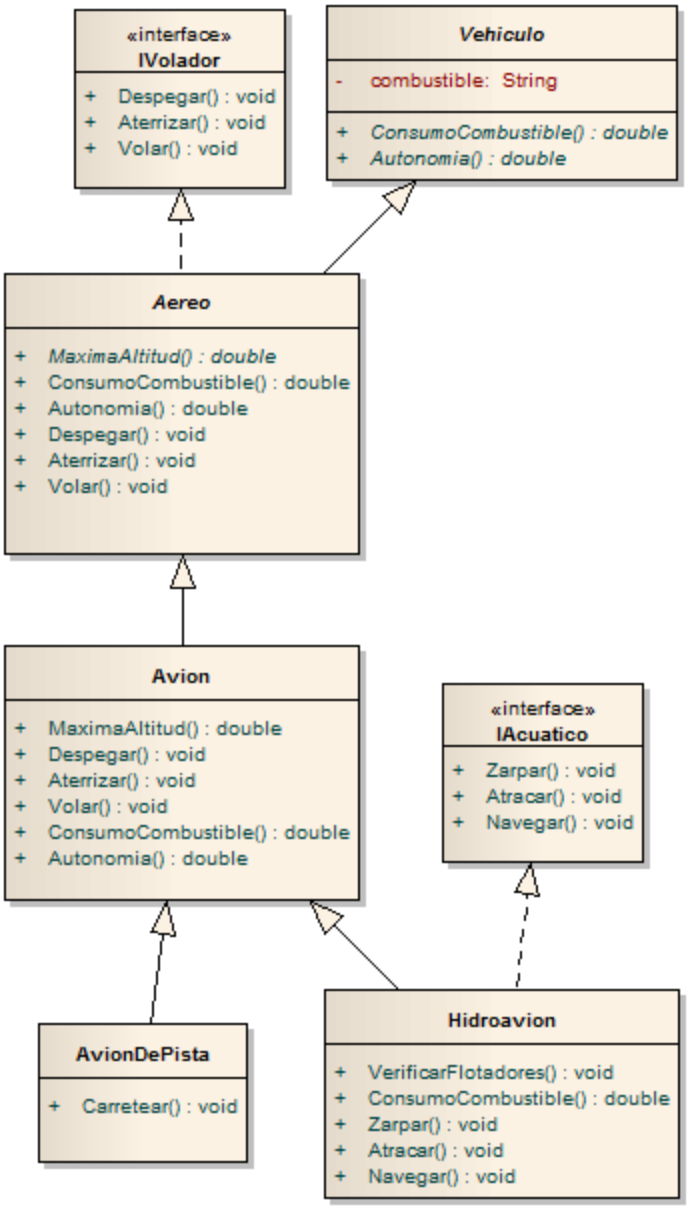
Otro detalle muy importante a tener en cuenta es que no sólo se extraen métodos al azar, sino aquellos que en conjunto puedan definir un concepto por si mismos (muchas veces esto sucede cuando un grupo de servicios son necesarios pero no lo suficientemente significativos en el contexto de un sistema para conformar un objeto por si mismo). En el ejemplo que se muestra a continuación, los métodos que se extraen son características inherentes de aquellos que son voladores, por eso se define la interfaz con el nombre Volador.

Es claro que el lugar que se implemente la interfaz no debe necesariamente ser donde se rescriban los métodos que esta define (recordar que los métodos de una interfaz son abstractos y deben rescribirse obligatoriamente para crear objetos de dicha clase), sino al menos, en una clase que la anteceda en la cadena de herencia.

En el ejemplo que se muestra a continuación, y sólo a modo de claridad en la exposición, se presenta la implementación de la interfaz al mismo nivel en el cuál se descubren los métodos con los que se compondrá la interfaz, pero esta claro que las clases que los rescribirán para convertirlos en métodos concretos, son sus subclases.

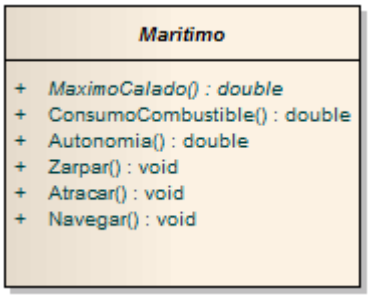
Sin embargo, en el caso de la clase Aéreo esto no afecta por ser abstracta, por lo tanto si no se sobrescriben los métodos no afecta su comportamiento porque antes de este refinamiento tampoco se podían crear objetos de este tipo.

Ejemplo



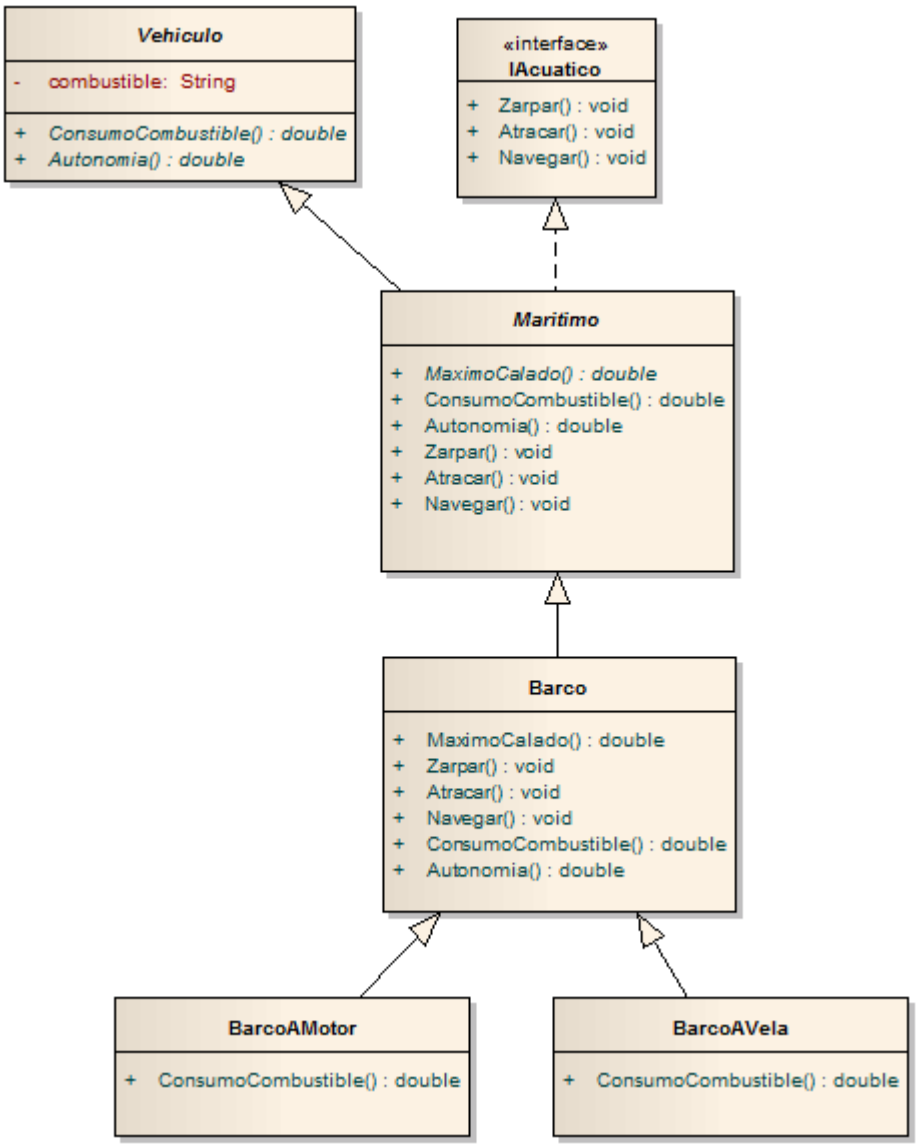
Al igual como se hizo en la sub cadena de herencia a partir de la clase Aereo, se puede tomar otra y realizar el mismo proceso. Por ejemplo, tomando a la clase Maritimo se puede realizar el mismo tipo de refinamiento

Ejemplo



Luego de extraer en una interfaz las acciones en común a otros objetos, el resultado es

Ejemplo



Partiendo de este último ejemplo, se puede ver como llevar al código el refinamiento del diseño. Por ejemplo, la declaración de la interfaz sería de la siguiente manera.

Ejemplo

C#

```
namespace transporteRefinamiento
{
    public interface IAcuatico
    {
        void Zarpar();
        void Atracar();
        void Navegar();
    }
}
```

VB

```
Public Interface IAcuatico
    Sub Zarpar()
    Sub Atracar()
    Sub Navegar()
End Interface
```

Notar que la implementación de la interfaz omite la declaración del modificador de visibilidad, lo cual no es un requerimiento en ella en C# y marca un error en VB.

La implementación de la misma en la clase abstracta.

Ejemplo

C#

```
namespace transporteRefinamiento
{
    public abstract class Maritimo : Vehiculo, IAcuatico
    {
        public abstract double MaximoCalado();

        public override double ConsumoCombustible()
        {
            return 0.0;
        }

        public override double Autonomia()
        {
            return 0.0;
        }

        public virtual void Zarpar()
        {
        }

        public virtual void Atracar()
```

```
        {
        }

        public virtual void Navegar()
        {
        }
    }
}
```

VB

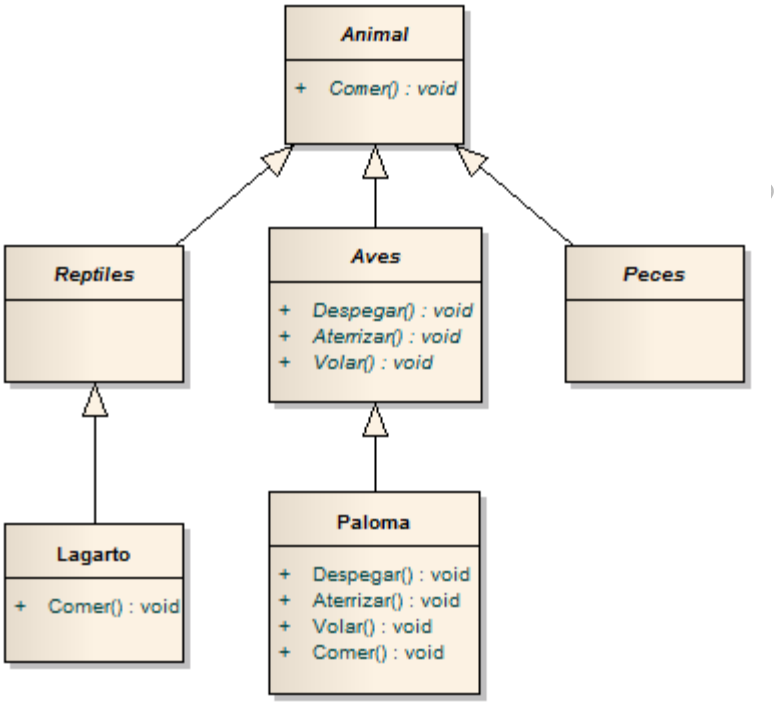
```
Public MustInherit Class Maritimo
    Inherits Vehiculo
    Implements IAcuatico

    Public MustOverride Function MaximoCalado() As Double

    Public Overrides Function Autonomia() As Double
        Return 0.0
    End Function

    Public Overrides Function ConsumoCombustible() As Double
        Return 0.0
    End Function

    Public Overridable Sub Atracar() Implements IAcuatico.Atracar

    End Sub

    Public Overridable Sub Navegar() Implements IAcuatico.Navegar

    End Sub

    Public Overridable Sub Zarpar() Implements IAcuatico.Zarpar

    End Sub
End Class
```

Notar que VB utiliza en los métodos de la interfaz `Implements IAcuatico.Zarpar`, lo cual es importante porque este lenguaje no es sensible al caso y esto provoca más ambigüedades que en C# que si lo es. De esta manera no hay duda que la implementación del método pertenece al declarado en la interfaz.

Para comprender como las interfaces pueden relacionar cadenas de herencia totalmente disímiles pero que comparten acciones en común, se plantea un nuevo conjunto de clases que pertenecen a cadenas de herencia que aparentemente no tiene nada en común y son totalmente diferentes de los ejemplos anteriores. El punto es demostrar como las interfaces consiguen formar verdaderos "puentes" entre clases para que se pueda utilizar comportamiento en común de los servicios que proveen.

El conjunto de clases expuesto pertenece a una representación de herencias de clases del tipo Animal.

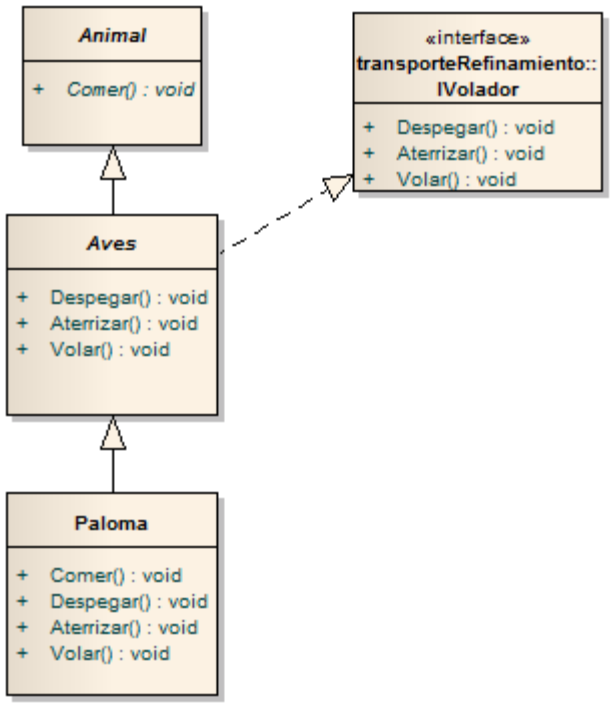
Ejemplo



Refinamiento

Si se observa con detenimiento la clase Aves, se puede realizar un refinamiento no sólo similar al del ejemplo anterior, sino que también se descubre que se puede reutilizar una interfaz ya definida, Volador. Repitiendo el mismo razonamiento que en el ejemplo de Aéreo, las clases quedarían con el siguiente formato

Ejemplo



Revisando el código, la interfaz IVolador tenía la siguiente forma.

Ejemplo

C#

```
namespace transporteRefinamiento
{
    public interface IVolador
    {
        void Despegar();
        void Aterrizar();
        void Volar();
    }
}
```

VB

```
Public Interface IVolador
    Sub Despegar()
    Sub Aterrizar()
    Sub Volar()
End Interface
```

Su posterior implementación en la clase obliga a importar la interfaz para su utilización en la clase.

Ejemplo

C#

```
using transporteRefinamiento;

namespace animalesRefinamiento
{
    public abstract class Aves : Animal, IVolador
    {
        public virtual void Despegar()
        {
        }

        public virtual void Aterrizar()
        {
        }

        public virtual void Volar()
        {
        }
    }
}
```

VB

```
Imports transporteRefinamiento

Public MustInherit Class Aves
    Inherits Animal
    Implements IVolador

    Public Overrides Sub Comer()

    End Sub

    Public Overridable Sub Aterrizar() Implements _
                    transporteRefinamiento.IVolador.Aterrizar

    End Sub

    Public Overridable Sub Despegar() Implements _
                    transporteRefinamiento.IVolador.Despegar

    End Sub

    Public Overridable Sub Volar() Implements transporteRefinamiento.IVolador.Volar

    End Sub
End Class
```

Nota: la clase Aves no define métodos propios a fines de simplificar el ejemplo

En la sub cadena de herencia a partir de la clase Aves, existen otras clases que tiene la obligación de sobrescribir todos los métodos abstractos para poder crear objetos de su tipo. Como la clase

Aves implementa la interfaz, los métodos de esta se suman a los definidos como abstractos en la clase como aquellos que se deben rescribir. La siguiente clase "concreta" demuestra el hecho con el método Comer, el cual es abstracto desde la clase Animal, y los métodos de la interfaz.

Ejemplo

C#

```
namespace animalesRefinamiento
{
    public class Paloma : Aves
    {
        public override void Comer()
        {
        }

        public override void Despegar()
        {
        }

        public override void Aterrizar()
        {
        }

        public override void Volar()
        {
        }
    }
}
```

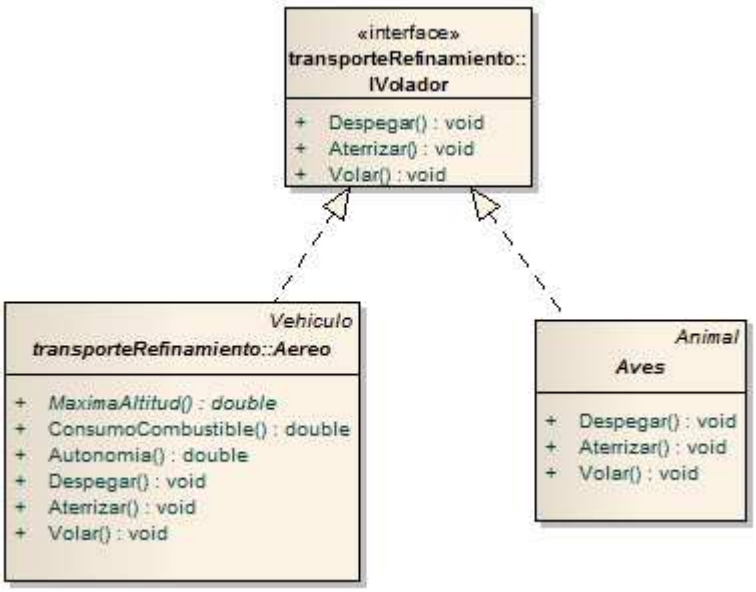
VB

```
Public Class Paloma
    Inherits Aves

    Public Overrides Sub Aterrizar()

    End Sub

    Public Overrides Sub Comer()

    End Sub

    Public Overrides Sub Despegar()

    End Sub

    Public Overrides Sub Volar()

    End Sub
End Class
```

Relacionando Refinamientos

Todo el proceso anterior permite la unión en un solo gráfico de las clases que implementan la interfaz para ver como se relacionan. Esto puede luego ser aprovechado por programas que quieran implantar el uso de dicha "relación".

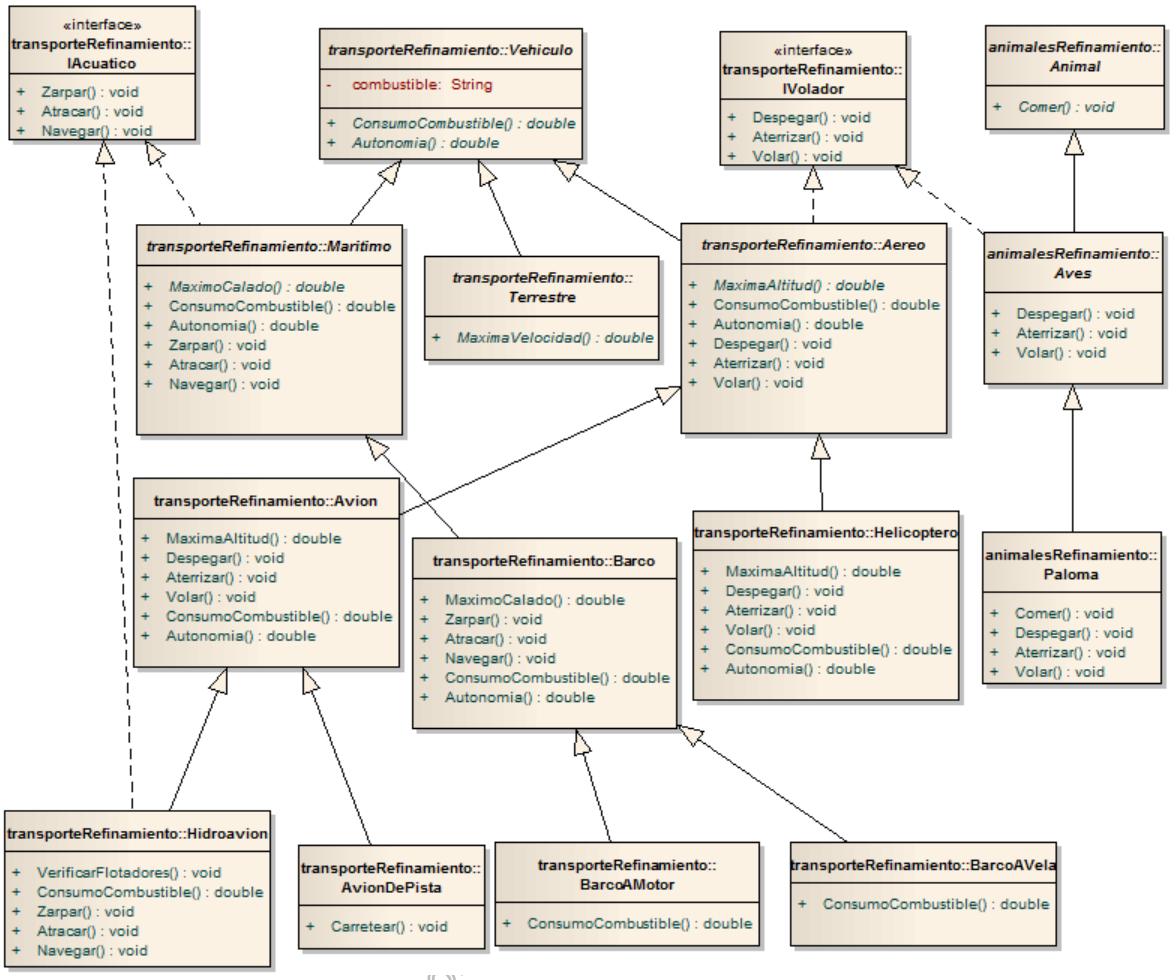
Ejemplo



Integración de Diagramas

Para comprender el formato final de los refinamientos y las cadenas de herencia que se verán afectadas por ellos, se realiza un diagrama que lo refleje. Este tipo de diagramas por lo general asiste en la definición de nuevas clases que utilicen los servicios que las interfaces definen.

Ejemplo



Uso de Interfaces

El diseño de interfaces permite la invocación de todos los métodos que las mismas definen en las clases concretas que los implementan.

Se debe tener cuidado al momento de diseñar la clase porque se pueden usar conceptualmente mal este tipo de relaciones.

El siguiente ejemplo muestra ese caso, en el cual una paloma **supuestamente** pide permiso para aterrizar y despegar. Si bien esto no tiene sentido, demuestra claramente el cuidado que se debe tener al aplicar las relaciones conseguidas por medio de las interfaces, porque el programa en si funciona perfectamente a nivel de código.

Nota: Tener en cuenta que los proyectos pueden y en este caso particularmente lo hacen, agregar referencia a otros proyectos. El uso de la importaciones demuestra los proyectos referenciados porque utilizan los espacios de nombres definidos en éstos

Ejemplo

C#

```
using transporteRefinamiento;
using animalesRefinamiento;

namespace interfaces
{
    class CampoDeAterrizaje
    {
        static void Main(string[] args)
        {
            CampoDeAterrizaje cda = new CampoDeAterrizaje();
            Helicoptero h = new Helicoptero();
            Hidroavion ha = new Hidroavion();
            Paloma p = new Paloma();
            p.Comer();
            Console.WriteLine("Autonomía del helicóptero");
            h.Autonomia();
            Console.WriteLine("Autonomía del hidroavión");
            ha.Autonomia();

            cda.PermisoParaAterrizar(h);
            cda.PermisoParaAterrizar(ha);
            cda.PermisoParaAterrizar(p);

            cda.PermisoParaDespegar(h);
            cda.PermisoParaDespegar(ha);
            cda.PermisoParaDespegar(p);
            Console.ReadKey();
        }

        public void PermisoParaAterrizar(IVolador v)
        {
            v.Aterrizar();
            //v.Autonomia(); Ilegal. No es accesible Autonomia()
        }
        public void PermisoParaDespegar(IVolador v)
        {
            v.Despegar();
        }
    }
}
```

VB

```
Imports transporteRefinamiento
Imports animalesRefinamiento

Public Class CampoDeAterrizaje
    Public Sub Inicio()
        Dim cda As New CampoDeAterrizaje()
        Dim h As New Helicoptero()
```

```
        Dim ha As New Hidroavion()
        Dim p As New Paloma()
        p.Comer()
        Console.WriteLine("Autonomía del helicóptero")
        h.Autonomia()
        Console.WriteLine("Autonomía del hidroavión")
        ha.Autonomia()

        cda.PermisoParaAterrizar(h)
        cda.PermisoParaAterrizar(ha)
        cda.PermisoParaAterrizar(p)

        cda.PermisoParaDespegar(h)
        cda.PermisoParaDespegar(ha)
        cda.PermisoParaDespegar(p)
        Console.ReadKey()
    End Sub

    Public Sub PermisoParaAterrizar(v As IVolador)
        v.Aterrizar()
        'v.Autonomia() Ilegal. No es accesible Autonomia()
    End Sub
    Public Sub PermisoParaDespegar(v As IVolador)
        v.Despegar()
    End Sub
End Class
```

Por otro lado, se puede apreciar en el siguiente ejemplo las ventajas brindadas por las interfaces al ser utilizadas correctamente.

Ejemplo

C#

```
using transporteRefinamiento;

namespace interfaces2
{
    public class Muelle
    {
        public static void Main(string[] args)
        {
            Muelle m = new Muelle();
            Hidroavion ha = new Hidroavion();
            BarcoAVela b = new BarcoAVela();

            m.PermisoParaAtracar(ha);
            m.PermisoParaAtracar(b);

            m.PermisoParaZarpar(ha);
            m.PermisoParaZarpar(b);
        }

        public void PermisoParaZarpar(IAcuatico a)
        {
```

```
            a.Zarpar();
        }
        public void PermisoParaAtracar(IAcuatico a)
        {
            a.Atracar();
        }
    }

}
```

VB

```
Imports transporteRefinamiento
Public Class Muelle
    Public Sub Inicio()
        Dim m As New Muelle()
        Dim ha As New Hidroavion()
        Dim b As New BarcoAVela()

        m.PermisoParaAtracar(ha)
        m.PermisoParaAtracar(b)

        m.PermisoParaZarpar(ha)
        m.PermisoParaZarpar(b)
    End Sub

    Public Sub PermisoParaZarpar(a As IAcuatico)
        a.Zarpar()
    End Sub
    Public Sub PermisoParaAtracar(a As IAcuatico)
        a.Atracar()
    End Sub
End Class
```

Nota: las interfaces definen la capacidad tecnológica de relacionar más de una clase de distintas cadenas de herencia, ***pero la responsabilidad de la implementación de los métodos y su posterior uso es del programador***. Las interfaces *por convención* tienen una **I** que antecede el nombre para ayudar a su visualización en el código.

Clases Anidadas

Cuando se crea una clase se puede declarar dentro de ella un atributo que sea referencia a un objeto. Esto permite implementar conceptos tan importantes como la agregación o la composición.

Sin embargo, existe otra posibilidad, la de declarar una clase dentro de otro tipo, sea este una clase o una estructura. Los lenguajes de .Net permiten que se declare una clase dentro del bloque de declaración de otro tipo y esto tiene consecuencias importantes respecto de las visibilidades que adquieren los elementos de ambas clases.

En primera instancia, parece que realizar esto no tiene mucho sentido porque la razón para hacerlo es que agrupa lógicamente clases que deben funcionar en conjunto (acoplamiento fuerte). La pregunta es, ¿por qué realizar esto si supuestamente podría hacerse con una composición?

La respuesta a esta pregunta radica en las capacidades declarativas de una clase y su fundamentación reside en las visibilidades, las capacidades de herencia única y la codificación asociadas a éstas. Por ejemplo, una clase, por más que se declare dentro de otra, tiene la capacidad de heredar e implementar tantas interfaces como quiera

Quedan por resolver las cuestiones de visibilidades de sus miembros. Por supuesto que se siguen respetando las reglas de bloques definidas con anterioridad, pero al utilizarlas en las invocaciones a elementos de la clase que anida a otra o creaciones de objetos del tipo de la clase que se encuentra anidada dentro de otra, se debe tener cuidado de tener bien definida la visibilidad para acceder.

Ejemplo

C#

```
namespace anidadas
{
    public class Exterior1
    {
        private int var=10;

        public int Var
        {
            get
            {
                return var;
            }
        }
        /* Declaración de una clase anidada llamada Anidada */
        internal class Anidada
        {
            private Exterior1 e;
            public Anidada(Exterior1 e)
            {
                this.e = e;
            }
            public Anidada():this(new Exterior1())
            {
            }
            public void HaceAlgo()
            {
                // La clase anidada tiene acceso a la variable var
                // de la clase exterior
                e.var=0;
                e.var++;
                Console.WriteLine(
                    "El valor de Exterior1.var en Exterior1.Anidada.haceAlgo es " +
```

```
                            e.Var);
          }
        }

        public void VerificaAnidada()
        {
            Anidada i = new Anidada();
            i.HaceAlgo();
        }
    }
}
```

VB

```
Public Class Exterior1
    Private _var As Integer = 10

    Public ReadOnly Property Var() As Integer
        Get
            Return _var
        End Get
    End Property

    ' Declaración de una clase anidada llamada Anidada
    Friend Class Anidada
        Private e As Exterior1
        Public Sub New(e As Exterior1)
            Me.e = e
        End Sub

        Public Sub New()
            MyClass.New(New Exterior1())
        End Sub

        Public Sub HaceAlgo()
            ' La clase anidada tiene acceso a la variable var
            ' de la clase exterior
            e._var = 0
            e._var += 1
            Console.WriteLine( _
              "El valor de Exterior1._var en Exterior1.Anidada.haceAlgo es " + _
                 e._var.ToString)
        End Sub
    End Class

    Public Sub verificaAnidada()
        Dim i As New Anidada()
        i.haceAlgo()
    End Sub

End Class
```

Independientemente de si el tipo externo es una clase o una estructura, los tipos anidados por defecto son privados, pero pueden ser `public`, `protected internal`, `protected`, `internal` y `private` o `Public`, `Protected Friend`, `Protected`, `Friend` y `Private` (C# o VB).

El tipo anidado o interior puede tener acceso al tipo contenedor o exterior. Para tener acceso al tipo contenedor, pasarlo como un constructor para el tipo anidado.

Visibilidades de las clases anidadas

Los tipos de nivel superior (clases que no se encuentran anidadas con otras), o sea, aquellos que no están anidados en otros tipos, sólo puede tener accesibilidad interna o pública. La accesibilidad por defecto para estos tipos es interna.

Los tipos anidados, que son miembros de otros tipos, pueden tener declaradas accesibilidades como se indica en la siguiente tabla.

Los miembros de	Acceso por defecto al miembro	Accesibilidad permitida en la declaración del miembro
`enum`	`public`	Ninguna
`class`	`private`	`public`
		`protected`
		`internal`
		`private`
		`protected internal`
`interface`	`public`	Ninguna
`struct`	`private`	`public`
		`internal`
		`private`

Propiedades de las Clases Anidadas

Las reglas para trabajar con clases anidadas están basadas en las propiedades que se definen a continuación:

- El nombre de la clase anidada deberá ser diferente al de la que la anida
- Se puede utilizar el nombre de la clase en una declaración sólo dentro del bloque en el que esta definida. Para usarla fuera de él se debe resolver visibilidad con un nombre calificado