Unidad 2

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 4

Clases

En este Capítulo

- Ensamblados (Assemblies)
- Ejecutable portable
- Declaraciones básicas
- Visibilidades: control de acceso a la clase
- Espacios de nombres
- Sobrecarga de Métodos
- Constructores sobrecargados
- ¿Cómo el Common Language Runtime realiza la recolección de basura?
- Los procesos de creación y destrucción de un objeto
- ¿Por qué utilizar el recolector de basura?
- Uso de this y Me
- Uso de this en Constructores (C#)
- Uso de Me.New en Constructores (VB)
- Herencia
- Niveles de acceso
- Los operadores de instancia

Ensamblados (Assemblies)

Son los componentes básicos de una aplicación en .NET. Esta consiste en uno o más ensamblados, y cada uno forma una unidad funcional. Un ensamblado puede ser un único ejecutable portable (PE – Portable Executable) como un exe. o una dll.

Cuando se crea un nuevo proyecto en Visual Studio se elige una plantilla o template del tipo de proyecto a crear. Muchos de estos tipos de proyectos se convierten en al menos un ejecutable portable (PE – Portable Executable). Cuando a un proyecto se le agregan referencias a otros proyectos dentro de una solución o se incluyen referencias a PE del Framework de .Net se pueden obtener varios PE como resultado de un mismo proyecto (PE se explica más adelante en este capítulo).

Alternativamente, un ensamblado puede contener varios archivos PE y archivos de otros recursos, tales como mapas de bits. Cada ensamblado tiene un manifiesto, que contiene metadatos sobre el ensamblado, como el título y la información de versión. Cuando se construye uno, el compilador toma el código fuente y lo traduce a código de lenguaje intermedio (IL). El just-in-time (JIT) luego traduce el código IL en código específico de la CPU cuando los usuarios ejecutar la aplicación.

Unidades de despliegue de ensamblados

Un ensamblado forma una unidad de despliegue. La ventaja de esta característica es que la aplicación puede cargar ensamblados sólo cuando se les requiere y descargar o recuperar ensamblados adicionales sobre demanda. Por lo tanto, la aplicación utiliza sólo los recursos que requiere.

Versiones de ensamblado

El ensamblado es la unidad más pequeña de la que se puede crear una versión. Las distintas versiones de un ensamblado pueden ejecutarse una al lado de la otra, y una aplicación cliente puede especificar qué versión de dichos ensamblados que se quiere usar. Esta capacidad elimina los conflictos que se pueden producir con aplicaciones que no utilicen el Framework de .NET.

Nota: Sólo se pueden versionar ensamblados que tienen un nombre fuerte.

Creación de un nombre seguro para un ensamblado

Un nombre seguro utiliza la criptografía de clave pública para identificar un ensamblado único. También utiliza una firma digital para autenticar el autor del ensamblado. Un nombre seguro consta de lo siguiente:

- El nombre del ensamblado.
- La versión del ensamblado.
- Opcionalmente, la información de referencia cultural para el ensamblado.
- Una clave pública.

El primer paso que se debe realizar para firmar un ensamblado con un nombre seguro es crear un par de claves criptográficas. Puede utilizar la herramienta para nombres seguros o fuertes (Sn.exe) para crear este par de claves. Sn.exe es parte del .NET Framework Software Development Kit (SDK).

Se puede utilizar esta utilidad abriendo una consola de comandos desde el menú de herramientas en la instalación del Visual Studio (Visual Studio Tools) y escribir el comando que se muestra en el siguiente ejemplo código para crear un par de claves.

```
sn –k NombreDeLaClave.snk
```

El segundo paso es firmar el ensamblado. Uno de los métodos que se puede utilizar para lograr esto es la herramienta Assembly Linker (Al.exe). Para utilizar la herramienta Al.exe para firmar un ensamblado con un nombre seguro, escribir luego de abrir una consola de comandos desde el menú de herramientas en la instalación del Visual Studio (Visual Studio Tools) el comando que se muestra en el siguiente ejemplo código para firmar un ensamblado con un nombre seguro.

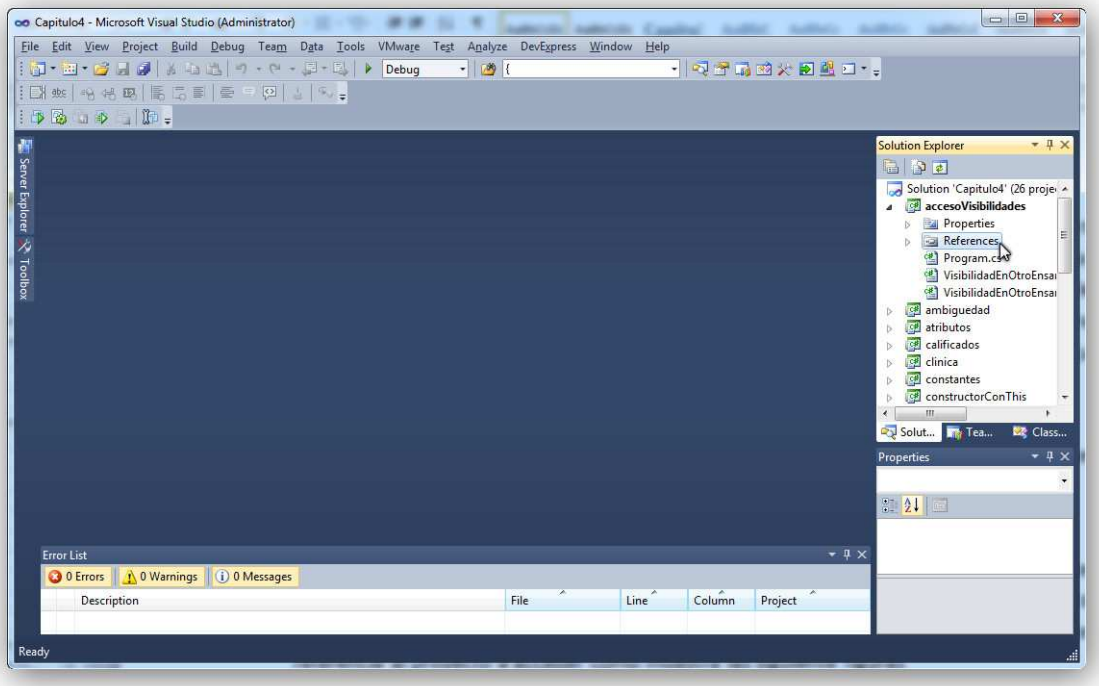
```
al /out:NombreDelEnsamblado.dll /keyfile:NombreDeLaClave.snk
```

También se puede realizar las mismas acciones desde las propiedades de un proyecto en la pestaña Firma (Signing).

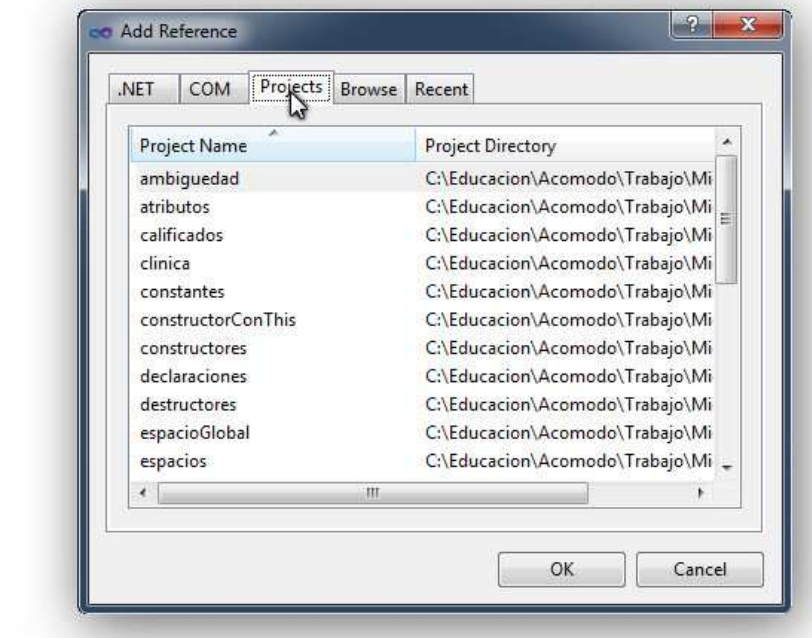
Agregar referencias en un proyecto para acceder a otro ensamblado

En Visual Studio se debe indicar cuando se quiere obtener acceso a los elementos declarados en otro proyecto. El acceso está definido a través del ensamblado que se crea cuando se compila (en versión final o de depuración) un proyecto. Para lograr dicho acceso, se debe especificar una referencia al proyecto a acceder como muestra las siguiente figuras.

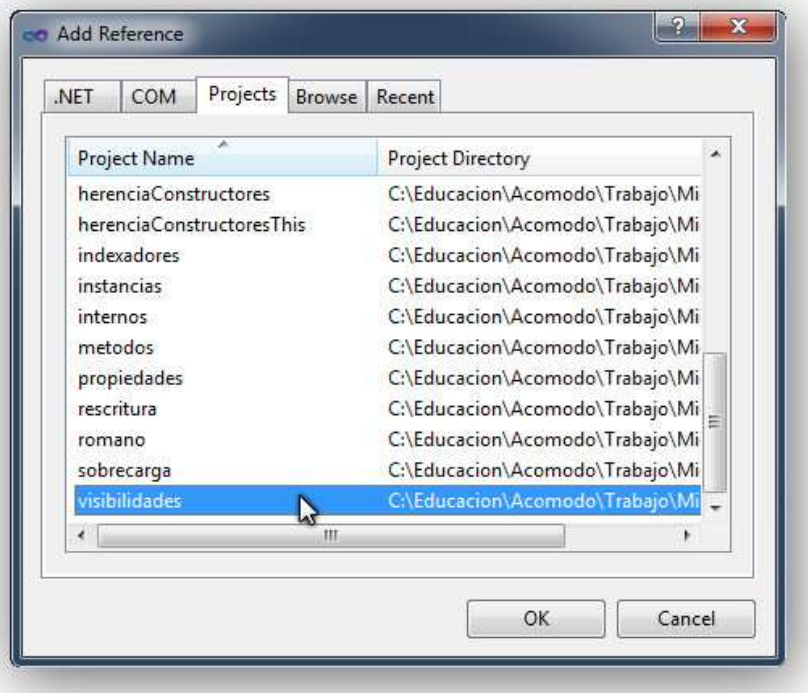
Todo proyecto posee una carpeta de referencias (para los ejemplos que se muestran se usa C# pero es análogo en VB) llamada References (en inglés, los entornos en castellano tiene nombres análogos traducidos) sobre la cual se debe realizar un clic derecho con el mouse:



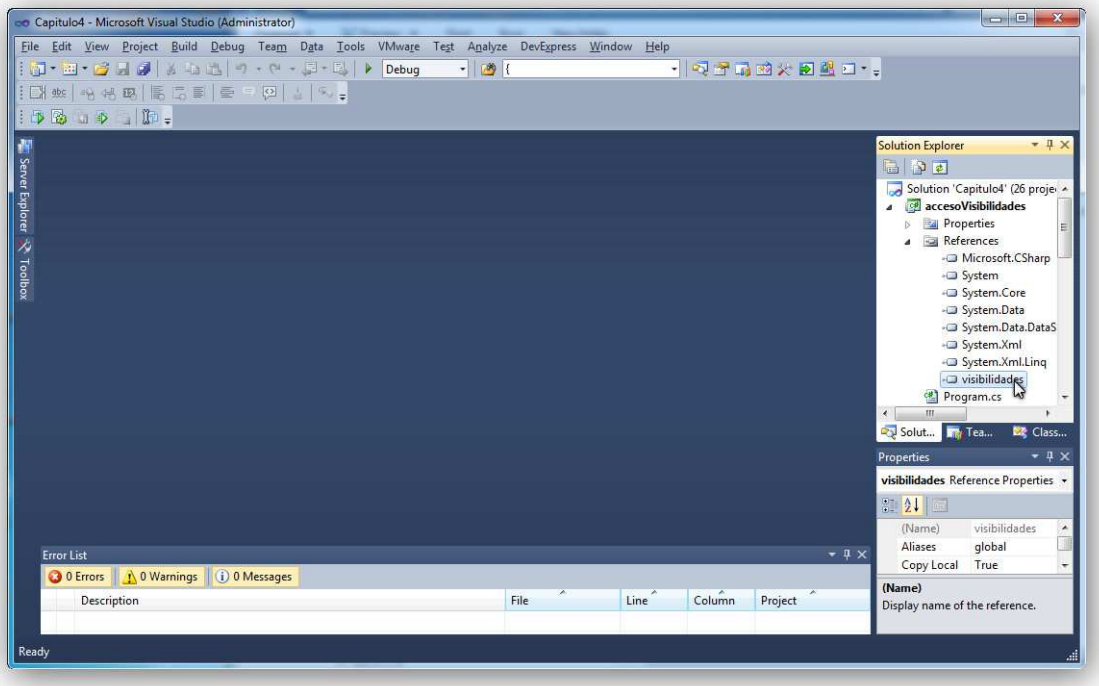
Al hacer clic derecho se despliega una ventana que permite seleccionar una referencia a un servicio o a un ensamblado. En este caso se busca hacer una referencia a un ensamblado de código instalado localmente, por lo tanto seleccionar Add Reference. Esto despliega la siguiente ventana.



Esta ventana permite crear referencia a distintos tipos de ensamblados, incluyendo aquellos provistos con el Framework de .Net. Sólo hay que seleccionar la pestaña deseada y elegir el ensamblado apropiado (como se puede observar, también permite añadir COM en formato de DLL). En el caso de referenciar otro proyecto, como es este, seleccionar la pestaña Projects para luego elegir el proyecto deseado (por ejemplo, se selecciona el proyecto visibilidades).



Esto determina que se agregue la referencia al proyecto visibilidades de manera de poder utilizar los elementos declarados en el. La referencia adicionada se puede ver en la carpeta References como se muestra a continuación.



Nota: Tener en cuenta que la referencia sólo brinda acceso. Una vez obtenido el mismo se deben utilizar los elementos dentro del proyecto por medio del acceso definido en su visibilidad como lo establece el lenguaje. Ver posteriormente en este capítulo espacios de nombres

Ejecutable portable

Metadatos

En el pasado, un componente de software (.exe o dll.) escrito en un lenguaje no podría utilizar un componente de software escrito en otro idioma. COM proporciona un paso adelante en la solución de este problema. El Framework .NET hace que sea aún más fácil la interoperación entre componentes al permitir que los compiladores emitan información declarativa adicional en todos los módulos y ensamblados. Esta información, llamada metadatos, ayuda a los componentes de interactuar sin problemas.

Metadatos y la estructura del archivo PE

Aunque la mayoría de los desarrolladores no necesitan conocer los detalles de implementación de metadatos, algunos podrían querer una comprensión más profunda. ***Estos conceptos son complementarios al conocimiento para desarrollar en .Net y no limitan la compresión para poder programar uno de sus lenguajes***.

Los metadatos se almacenan en una sección de un archivo ejecutable portable (PE) del Framework de .NET mientras que lo referente al lenguaje intermedio de Microsoft (MSIL) se almacena en otra

sección del archivo PE. La parte de los metadatos del archivo contiene de una serie de tablas y amontonamientos de estructuras de datos. La parte de MSIL contiene MSIL y cadenas identificadoras de metadatos que hacen referencia a la parte de metadatos del archivo PE. Se pueden encontrar las cadenas identificadoras de metadatos al utilizar herramientas como el desensamblador de MSIL (Ildasm.exe) para ver el código MSIL o el depurador en tiempo de ejecución (Cordbg.exe) para realizar un volcado de memoria.

Las tablas de metadatos y los amontonamientos

Cada tabla de metadatos contiene información acerca de los elementos de su programa. Por ejemplo, una tabla de metadatos describe las clases en el código, en otra tabla se describen los atributos y así sucesivamente. Si se tienen diez clases en el código, la tabla de clases tendrá 10 filas, una para cada clase. Las tablas de metadatos referencian otras tablas y amontonamientos. Por ejemplo, la tabla de metadatos para clases referencia la tabla de métodos.

Los metadatos también almacenan información en cuatro estructuras de amontonamientos: cadenas, BLOBs (Binary Large Object), cadenas del usuario y GUID (Globally Unique Identifier). Todas las cadenas usadas para nombrar tipo y miembros se almacenan en el amontonamiento de cadenas. Por ejemplo, una tabla de métodos no almacena directamente el nombre de un método en particular, sino que apunta al nombre del método almacenado en el amontonamiento de cadenas.

Cadenas identificadoras de metadatos

Cada fila de cada tabla de metadatos se identifica unívocamente en la parte de MSIL del archivo PE por una cadena identificadora de metadatos. Estas cadenas son conceptualmente similares a los punteros, persistidas (almacenamiento estable, como por ejemplo, en el disco rígido) en MSIL, que hacen referencia a una tabla de metadatos particular.

Una cadena identificadora de metadatos es un número de cuatro bytes. El byte superior denota la tabla de metadatos a la que la cadena en particular se refiere (métodos, tipo y así sucesivamente). Los tres bytes restantes especifican la fila en la tabla de metadatos que se corresponde con el elemento de programación que se describe. Si se define un método en C# y se lo compila en un archivo PE, la siguiente cadena identificadora de metadatos puede, por ejemplo, existir en la parte de MSIL del archivo PE:

0x06000004

El byte superior (0x06) indica que este es una cadena identificadora **MethodDef**. Los tres bytes más bajos (000004) le indican al Common Language Runtime que busque en la cuarta fila de la tabla **MethodDef** la información que describe la definición de este método.

Los metadatos en un archivo PE

Cuando se compila un programa para el CLR, se convierte en un archivo PE que consta de tres partes. La tabla siguiente describe el contenido de cada parte.

Sección del PE	Contenido de la sección del PE
Cabecera del PE	El índice de las secciones principales del archivo PE y la dirección del punto de entrada. El CLR en tiempo de ejecución utiliza esta información para identificar el archivo como uno del tipo PE y para determinar dónde se inicia la ejecución cuando se carga el programa en la memoria.
Instrucciones MSIL	Las instrucciones en lenguaje intermedio de Microsoft (MSIL) que componen el código. Muchas instrucciones MSIL se acompañan con cadenas identificadoras de metadatos.
Metadatos	Tablas de metadatos y amontonamientos. El CRL en tiempo de ejecución utiliza esta sección para registrar información sobre todos los tipos y miembros en el código. Esta sección también incluye los atributos personalizados e información de seguridad.

Declaraciones básicas

Las clases son el mecanismo por el que se pueden crear nuevos tipos, tanto en C# como en VB. Las clases son el punto central sobre el que giran la mayoría de los conceptos de la Orientación a Objetos.

Las clases, al igual que las estructuras de datos en lenguajes como C, son tipos de datos definidos por el usuario, por lo tanto, el compilador no sabe nada de ellos hasta que se define la forma que tendrán.

La declaración de la forma de los tipos de datos definidos por el usuario se especifica en una clase y cuando se crea una variable del tipo de dicha clase declarada, se crea un objeto.

Las clases se componen principalmente de métodos y atributos que la definen. Los primeros son los servicios que los objetos de ese tipo brindarán. Los atributos, en cambio, tendrán almacenados los valores que describen a ese objeto en particular y sobre los cuales actuarán los métodos.

Ejemplo

C#

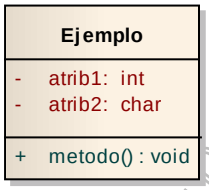
```
namespace declaraciones
{
    class Ejemplo
    {
        private int atrib1;
```

```
        private char atrib2;
        public void metodo() { }
    }
}
```

VB

```
Public Class Ejemplo
    Private atrib1 As Integer
    Private atrib2 As Char
    Public Sub metodo()
    End Sub
End Class
```

El lenguaje de modelización UML permite realizar una descripción gráfica de la clase de la siguiente forma:



Los componentes de una clase

Dentro del cuerpo de una clase se pueden incluir declaraciones diferentes, entre las que se encuentran:

- Constantes
- Atributos
- Métodos
- Constructores
- Destructores
- Propiedades
- Eventos
- Indexadores
- Operadores
- Otros tipos (clases)

Las constantes

Como su nombre lo indica, son valores inalterables en el transcurso de un programa. Para declararlas se debe anteponer la palabra clave `const` o `Const` (C# o VB) al tipo de datos que define el valor de la constante.

Ejemplo

C#

```
private const double PI = 3.1416;
```

VB

```
Private Const PI As Double = 3.1416
```

El compilador remplazará a lo largo del código el identificador PI por el valor asignado a la constante, lo cual permite optimizar espacios de almacenamiento interno en memoria por no ser una variable.

El beneficio de su utilización radica en darle un nombre mnemotécnico a un valor constantes que si se desea cambiar en el futuro la modificación afecte sólo el lugar donde se realizó la declaración y no a lo largo del código donde se utilice dicho valor. Por ejemplo, si se quiere declarar un valor para la constante Pi, y luego se desea agregar más dígitos decimales al valor, en lugar de cambiar todos los lugares en los cuales se utilizó, sólo se cambia la declaración de la constante utilizada.

Ejemplo

C#

```
namespace constantes
{
    class Circulo
    {
        private const double PI = 3.1416;

        public double CalculaArea(double radio)
        {
            return radio * PI * PI;
        }

        public double CalculasPerimetro(double radio)
        {
            return 2 * radio * PI;
        }
    }
}
```

VB

```
Public Class Circulo
    Private Const PI As Double = 3.1416

    Public Function CalculaArea(ByVal radio As Double) As Double
        Return radio * PI * PI
    End Function

    Public Function CalculasPerimetro(ByVal radio As Double) As Double
        Return 2 * radio * PI
    End Function
```

```
End Class
```

Los atributos

Los atributos son las variables de instancia de la clase, los cuales son espacios de almacenamiento reservados en tiempo de ejecución en el heap. Estos **mantienen el estado de un objeto** en tiempo de ejecución. Pueden tener un valor inicial y pueden ser declarados de solo lectura con la palabra clave `readonly` o `ReadOnly` (C# o VB). Cuando un atributo (muchas veces llamado **campo**) es de sólo lectura, se le puede asignar un valor únicamente en el constructor. A partir de ese punto se comporta como una constante a pesar de no serlo. La principal diferencia entre los atributos de sólo lectura y las constantes es que estas últimas deben tener su valor declarado al momento de compilar la clase, mientras que los atributos de sólo lectura pueden además recibir sus valores en tiempo de ejecución y por lo tanto nunca pueden ser constantes. Cualquier cambio que se intente hacer a un atributo `readonly` o `ReadOnly` (C# o VB) en tiempo de ejecución generará un error en tiempo de compilación como el siguiente:

```
A readonly field cannot be assigned to (except in a constructor or a variable
initializer)
```

Los atributos tienen un `this` o `Me` (C# o VB) asociado tiempo de ejecución (esto es, cuando se cree un objeto con la clase que lo contiene) porque son parte de un objeto en tiempo de ejecución salvo que sean estáticos (lo cual se verá posteriormente) y se lo puede utilizar con notación de punto para invocarlo siempre y cuando sean visibles para usarlos de esta manera.

Los métodos

Los métodos son los servicios que presta una clase. Hasta el momento se los definió pero no se detalló algunas características importantes de los mismos. Se debe tener en cuenta algunos detalles importantes respecto de los métodos que pertenecen a una clase:

- Toda variable declarado en un método se considera local al mismo y debe estar inicializada antes de poder ser utilizada en cualquier tipo de operación
- Los métodos tienen un `this` o `Me` (C# o VB) asociado tiempo de ejecución (esto es, cuando se cree un objeto con la clase que lo contiene) porque son parte de un objeto en tiempo de ejecución salvo que sean estáticos (lo cual se verá posteriormente) y se lo puede utilizar con notación de punto para invocarlo siempre y cuando sean visibles para usarlos de esta manera.
- Los métodos se pueden invocar desde cualquier parte dentro del cuerpo de una clase sin necesidad de la notación de punto.
- Un método puede recibir uno o más parámetros. Los mismos están clasificados según su tipo, como se detallará posteriormente, lo cual define el comportamiento del parámetro dentro del método.
- Los métodos definen el comportamiento de un objeto en tiempo de ejecución. Esto es, indican las acciones que se pueden realizar con un objeto del tipo de la clase que lo define.

Los parámetros de los métodos se pueden clasificar como:

Parámetros de entrada: siempre se pasan por valor. El valor almacenado por un parámetro de este tipo sólo es accesible durante la ejecución del método. Su lugar de almacenamiento es el lugar destinado en el stack que pertenece al método. Cuando se pasa como argumento una variable o literal de asignación, el valor se copia en el parámetro del método en el stack que le pertenece.

Parámetros de salida: Los parámetros de salida son parámetros cuyos valores no se establecen cuando se llama al método. En su lugar, el método establece los valores y devuelve los valores a la función que llama a través del parámetro de salida. Esto implica que los parámetros de salida pueden inclusive ser tipos que almacenen `null` o `Nothing` (C# o VB). Los parámetros de salida se especifican en las listas de parámetros con la palabra clave `out` (sólo en C#). Dicha palabra clave debe preceder al tipo declarado de parámetro en la lista de parámetros. Cuando se llama a un método con un parámetro de salida, se debe declarar una variable que contenga al valor que éste asumirá.

Parámetros por referencia: Estos parámetros proporcionan valores por referencia. En otras palabras, el método recibe una referencia a la variable especificada cuando se lo llama, por lo tanto el espacio de almacenamiento sobre el cual se trabaja es el del método que realizó la llamada e invocó al que recibe la referencia. No se debe confundir un argumento por referencia con el pasaje de argumento de objetos. Si bien estos últimos son también referencias, las mismas apuntan a un objeto almacenado en el heap. Los parámetros por referencia se especifican en listas de parámetros con la palabra clave `ref` o `ByRef` (C# o VB). Dicha palabra clave debe preceder al tipo del parámetro en la lista de parámetros. También se debe colocar en el método que realiza la invocación.

Vector de parámetros: Existen situaciones en las cuales que un método no conozca cuántos parámetros debe aceptar al ser diseñado. El vector de parámetros resuelve este problema de diseño ya que permite especificar que el método acepte un número variable de argumentos. Se declara usando la palabra clave `params` o `ParamArray` (C# o VB), seguida del tipo de variable que deberá proporcionar el método que realiza el llamado. La especificación de tipos va seguida de corchetes o paréntesis (C# o VB), que a su vez van seguidos del identificador del vector de parámetros

Ejemplo

C#

```
namespace metodos
{
    class Parametros
    {
        private void PruebaRef(ref int uno, int dos)
        {
            // Esta asignación afectará al parámetro
```

```
        uno += dos;
        // Esta no afectará al valor usado como segundo argumento
        dos = 999;
    }

    public void ProbandoRef()
    {
        int uno = 5;
        int dos = 2;
        Console.WriteLine("Prueba de ref");
        Console.WriteLine("uno= {0}, dos = {1}", uno, dos);
        PruebaRef(ref uno, dos);
        Console.WriteLine("uno= {0}, dos = {1}", uno, dos);
    }

    private void PruebaOut(out int uno)
    {
        uno = 1;
    }

    public void ProbandoOut()
    {
        int valor = 0;
        Console.WriteLine("Prueba de out");
        Console.WriteLine("La variable valor antes de la llamada " +
            "no está inicializada y no se puede usar.");
        PruebaOut(out valor);
        Console.WriteLine("Variable valor después de la llamada: " + valor);
    }

    public void PruebaPasajePorValor(int uno)
    {
        uno = 15;
    }

    public void ProbandoPasajePorValor()
    {
        int valor = 15;
        Console.WriteLine("Prueba de pasaje por valor");
        Console.WriteLine("Variable valor antes de la llamada: " + valor);
        PruebaPasajePorValor(valor);
        Console.WriteLine("Variable valor después de la llamada: " + valor);
    }

    public void PruebaParamsConTipoPorValor(params int[] lista)
    {
        for (int i = 0; i < lista.Length; i++)
            Console.WriteLine(lista[i]);
        Console.WriteLine();
    }

    public void PruebaParamsConObjetos(params object[] lista)
    {
        for (int i = 0; i < lista.Length; i++)
            Console.WriteLine(lista[i]);
```

```
            Console.WriteLine();
        }

        public void ProbandoParams()
        {
            Console.WriteLine("Prueba de params");
            PruebaParamsConTipoPorValor(1, 2, 3);
            PruebaParamsConObjetos(1, 'a', "test");

            int[] vec = new int[3] { 10, 11, 12 };
            PruebaParamsConTipoPorValor(vec);
        }
    }
}
```

VB

```
Public Class Parametros
    Private Sub PruebaRef(ByRef uno As Integer, dos As Integer)
        ' Esta asignación afectará al parámetro
        uno += dos
        ' Esta no afectará al valor usado como segundo argumento
        dos = 999
    End Sub

    Public Sub ProbandoRef()
        Dim uno As Integer = 5
        Dim dos As Integer = 2
        Console.WriteLine("Prueba de ref")
        Console.WriteLine("uno= {0}, dos = {1}", uno, dos)
        PruebaRef(uno, dos)
        Console.WriteLine("uno= {0}, dos = {1}", uno, dos)
    End Sub

    Public Sub PruebaPasajePorValor(uno As Integer)
        uno = 15
    End Sub

    Public Sub ProbandoPasajePorValor()
        Dim valor As Integer = 15
        Console.WriteLine("Prueba de pasaje por valor")
        Console.WriteLine("Variable valor antes de la llamada: " + valor.ToString)
        PruebaPasajePorValor(valor)
        Console.WriteLine("Variable valor después de la llamada: " + valor.ToString)
    End Sub

    Public Sub PruebaParamArrayConTipoPorValor(ParamArray lista() As Integer)
        For i As Integer = 0 To lista.Length - 1
            Console.WriteLine(lista(i))
            Console.WriteLine()
        Next
    End Sub

    Public Sub PruebaParamArrayConObjetos(ParamArray lista() As Object)
        For i As Integer = 0 To lista.Length - 1
            Console.WriteLine(lista(i))
```

```
            Console.WriteLine()
        Next
    End Sub

    Public Sub ProbandoParamArray()
        Console.WriteLine("Prueba de ParamArray")
        PruebaParamArrayConTipoPorValor(1, 2, 3)
        PruebaParamArrayConObjetos(1, "a"c, "test")

        Dim vec() As Integer = {10, 11, 12}
        PruebaParamArrayConTipoPorValor(vec)
    End Sub
End Class
```

Si se ejecuta el siguiente programa

C#

```
namespace metodos
{
    class Program
    {
        static void Main(string[] args)
        {
            Parametros p = new Parametros();

            p.ProbandoPasajePorValor();
            p.ProbandoOut();
            p.ProbandoRef();
            p.ProbandoParams();
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
    Sub Main()
        Dim p As New Parametros()

        p.ProbandoPasajePorValor()
        p.ProbandoRef()
        p.ProbandoParamArray()
        Console.ReadKey()
    End Sub
End Module
```


Produce la siguiente salida en C# y una salida similar en VB sin la las tres líneas a partir de “prueba de out” inclusive.

```
Prueba de pasaje por valor
Variable valor antes de la llamada: 15
Variable valor después de la llamada: 15
Prueba de out
```

```
La variable valor antes de la llamada no está inicializada y no se puede usar.
Variable valor después de la llamada: 1
Prueba de ref
uno= 5, dos = 2
uno= 7, dos = 2
Prueba de params
1
2
3

1
a
test

10
11
12
```

Los constructores

Todas las clases en .Net tienen métodos especiales llamados constructores que se utilizan para inicializar un objeto nuevo de ese tipo. Se puede declarar e implementar un constructor como se haría con cualquier otro método en una clase, salvo por un detalle, no se tiene que especificar el valor de retorno del constructor porque esta predefinido (devuelve la referencia al objeto que se crea).

Los constructores tienen el mismo nombre que la clase en C# y se llaman `New` en VB, por ejemplo, el nombre del constructor de la clase Rectangulo es Rectangulo(), el nombre del constructor de la clase Circulo es Circulo(), etc...

Ejemplo

C#

```
namespace constructores
{
    class Ejemplo
    {
        private int atrib1;
        private char atrib2;
        public Ejemplo(int a) { atrib1 = a; }
        public void metodo()
        {
            //[sentencias;]
        }
    }
}
```

VB

```
Public Class Ejemplo
    Private atrib1 As Integer
    Private atrib2 As Char
    Public Sub New(ByVal a As Integer)
        atrib1 = a
    End Sub

    Public Sub metodo()
        '[sentencias]
    End Sub
End Class
```

La clase Ejemplo no tiene constructor por defecto. Si se quiere diseñar la clase para que lo posea, se debe declarar explícitamente.

Las clases tienen siempre un constructor explícito o implícito. Si no se declara uno explícitamente, el lenguaje, C# o VB, lo agrega sin argumentos, lo cual quiere decir que si no se agrega la definición un constructor sería como si la clase tuviera declarado en su interior.

C#

```
public Ejemplo()
```

VB

```
Public Sub New()
```

Si el constructor es explícito debe tener el mismo nombre de la clase en C# o llamarse `New` en VB y no retornar ningún valor, pero cuando se agrega un constructor explícito, el lenguaje no agrega el constructor por defecto (aquel que no recibe ningún parámetro), lo que implica para tenerlo que se debe declarar en el código.

Ejemplo

C#

```
namespace constructores.defecto
{
    class Ejemplo
    {
        private int atrib1;
        private char atrib2;
        public Ejemplo(int a) { atrib1 = a; }
        public Ejemplo() { atrib1 = 5; }
        public void metodo()
        {
            //[sentencias;]
        }
    }
}
```

VB

```
Namespace defecto
```

```
    Public Class Ejemplo
        Private atrib1 As Integer
        Private atrib2 As Char

        Public Sub New(ByVal a As Integer)
            atrib1 = a
        End Sub

        Public Sub New()
            atrib1 = 5
        End Sub

        Public Sub metodo()
            '[sentencias]
        End Sub
    End Class
End Namespace
```

Cuando se declaren constructores para las clases, se pueden utilizar los modificadores de acceso que determinan si otros objetos pueden crear otros de su tipo (los modificadores se muestran primero en C# y luego en VB):

- `private` o `Private`
  - Ninguna otra clase puede crear un objeto de su clase. La clase puede contener métodos públicos estáticos y esos métodos pueden construir un objeto y devolverlo, pero nada más. Los métodos estáticos se explicarán posteriormente.
- `protected` o `Protected`
  - Sólo las subclases de la clase o aquellas que se encuentren en el mismo paquete pueden crear objeto de ella.
- `public` o `Public`
  - Cualquiera pueda crear un objeto de la clase.
- `internal` o `Friend`
  - Nadie externo al ensamblado puede construir un objeto de su clase. Esto es muy útil si se quiere que las clases que tenemos en un ensamblado puedan crear objetos de la clase pero no se quiere que lo haga nadie más.

.Net soporta poner más de un constructor en una clase gracias a la sobrecarga de los nombres de métodos, tema que se explicará posteriormente. Una clase puede tener cualquier número de constructores, todos los cuales tienen el mismo nombre. Si este es el caso, los mismos deben diferenciarse unos de otros cumpliendo al menos una de las siguientes condiciones en sus argumentos:

- El número
- El tipo (para la misma posición de un argumento en la lista)

La clase Ejemplo a continuación, proporciona dos constructores diferentes, ambos llamados Ejemplo(), pero cada uno con número o tipo diferentes de argumentos a partir de los cuales se puede crear un nuevo objeto Ejemplo. Particularmente en este caso, sólo cambia el tipo de argumento

Ejemplo

C#

```
namespace constructores.sobrecargados
{
    class Ejemplo
    {
        private int atrib1;
        private char atrib2;
        public Ejemplo(int a) { atrib1 = a; }
        public Ejemplo(char a) { atrib2 = a; }
        public void metodo()
        {
            //[sentencias;]
        }
    }
}
```

VB

```
Namespace sobrecargados
    Public Class Ejemplo
        Private atrib1 As Integer
        Private atrib2 As Char

        Public Sub New(ByVal a As Integer)
            atrib1 = a
        End Sub

        Public Sub New(ByVal a As Char)
            atrib2 = a
        End Sub

        Public Sub metodo()
            '[sentencias]
        End Sub
    End Class
End Namespace
```


Un constructor utiliza sus argumentos para inicializar el estado del nuevo objeto. Entonces, cuando se crea un objeto, se debe elegir el constructor cuyos argumentos reflejen mejor cómo se quiere inicializar dicho objeto.

Basándose en el número y tipos de los argumentos que se pasan al constructor, el compilador determina cual de ellos utilizar, Así el compilador sabe que cuando se escribe:

C#

```
constructores.sobrecargados.Ejemplo e3 = new
                constructores.sobrecargados.Ejemplo(10);
```

Utilizará el constructor que requiere un argumento entero, y cuando se escribe:

C#

```
constructores.sobrecargados.Ejemplo e3 = new
                constructores.sobrecargados.Ejemplo('a');
```

Utilizará el constructor que requiere como argumento un carácter.

Los destructores

Las clases pueden definir un destructor, que es un método especial que se ejecuta cuando el CLR destruye los objetos de la clase. Se pueden pensar a los destructores como lo contrario de los constructores. Estos se ejecutan cuando se crean objetos y los destructores se ejecutan cuando el recolector de basura (Garbage Collector) destruye los objetos no utilizados. Este proceso se ejecuta independientemente sin la intervención del programador.

Los destructores son opcionales. Es perfectamente válido escribir una clase sin un destructor (y hasta ahora, es lo que se ha estado haciendo en los ejemplos). Si se escribe un destructor, sólo se puede escribir uno.

> **Nota**: A diferencia de los constructores, no se puede definir más de un destructor para una clase.

Los destructores se escriben de la siguiente forma:

- El símbolo de tilde (~).
- El identificador del destructor, que debe ser el mismo nombre que el de la clase.
- Un conjunto de paréntesis. Los constructores **nunca** reciben argumentos.
- Los constructores **nunca** retornan un valor.

> **Nota**: Recordar que un objeto es seleccionable para el recolector de basura cuando no tiene ninguna referencia válida en el código que se esta ejecutando que apunte a él. Esto es, ninguna variable de referencia tiene almacenada su dirección en el heap en ningún lugar del código que se está ejecutando

Las propiedades

Las propiedades son miembros identificados que proporcionan acceso al estado de un objeto. Las propiedades tienen un tipo, un identificador y tienen uno o dos fragmentos de código asociados a ellos: un método `get` o `Get` (C# o VB) y un método `set` o `Set` (C# o VB). Dichos métodos reciben el nombre de descriptores de acceso o mutación respectivamente. Cuando un cliente accede a una propiedad se ejecuta el descriptor de acceso `get` o `Get` (C# o VB) de la propiedad. Cuando el

cliente establece un nuevo valor para la propiedad se ejecuta el descriptor de mutación `set` o `Set` (C# o VB) de la propiedad.

Como los descriptores son esencialmente métodos, se pueden escribir dentro de ellos instrucciones que permiten validar o establecer rangos de valores admitidos por las propiedades. Esto es especialmente útil para controlar el valor en las variables de instancia, para implementar reglas de negocios sobre las mismas y principalmente para poner en práctica el principio de ocultamiento de la información.

Otra característica interesante es que la invocación a los métodos `set` y `get` o `Set` y `Get` (C# o VB) de las propiedades se oculta al programador el cuál la utiliza como si fuera una asignación de variable a través del nombre de la misma, obteniendo el valor que almacena o cambiándolo como si fuera una variable local.

Ejemplo

C#

```
namespace propiedades
{
    class Punto
    {
        private double x;
        private double y;

        public Punto(double x, double y)
        {
            this.x = x;
            this.y = y;
        }
        public double X
        {
            get
            {
                return x;
            }
            set
            {
                x = value;
            }
        }
        public double Y
        {
            get
            {
                return y;
            }
            set
            {
                y = value;
            }
```

```
        }
    }
}
```

VB

```
Public Class Punto
    Private _x As Double
    Private _y As Double

    Public Sub New(x As Double, y As Double)
        _x = x
        _y = y
    End Sub
    Public Property X() As Double
        Get
            Return _x
        End Get
        Set(ByVal value As Double)
            _x = value
        End Set
    End Property
    Public Property Y() As Double
        Get
            Return _y
        End Get
        Set(ByVal value As Double)
            _y = value
        End Set
    End Property

End Class
```

El siguiente programa utiliza las propiedades de la clase Punto

C#

```
namespace propiedades
{
    class Program
    {
        static void Main(string[] args)
        {
            Punto p = new Punto(3.3, 4.4);
            p.X = 5.5;
            p.Y = 6.6;
            Console.WriteLine("El valor X del punto es: " + p.X);
            Console.WriteLine("El valor Y del punto es: " + p.Y);
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
```

```
    Sub Main()
        Dim p As New Punto(3.2999999999999998, 4.4000000000000004)
        p.X = 5.5
        p.Y = 6.5999999999999996
        Console.WriteLine("El valor X del punto es: " + p.X.ToString)
        Console.WriteLine("El valor Y del punto es: " + p.Y.ToString)
        Console.ReadKey()
    End Sub

End Module
```

Nota: los números constantes en el programa de VB los coloca automáticamente el entorno. Esto se debe a las aproximaciones que posee VB con los números de punto flotante

La salida producida es:

El valor X del punto es: 5,5
El valor Y del punto es: 6,6

Las propiedades pueden ser de sólo escritura o lectura. En el caso de C# sólo se debe poner el descriptor adecuado en cada caso. Sin embargo VB necesita que se declare explícitamente la propiedad cuando es de sólo lectura como `ReadOnly`

Los eventos

.Net permite que las clases informen a otras partes del código cuando se produce una acción sobre ella. Esta capacidad recibe el nombre de mecanismo de evento y permite informar a los elementos que manejan al evento si se produce uno en la clase. Esto se logra derivando el procesamiento al lugar del código especificado como el manejador del evento (el cual es un método como cualquier otro). Se puede diseñar clases que informen a otros fragmentos de código cuando se produzcan determinados eventos en la clase que "dispara" al mismo. Quizás se quiera usar un evento para informar a otros fragmentos de código de que se ha completado una operación muy larga. Por ejemplo, si se quiere diseñar una clase que lea un archivo de disco de gran tamaño. Si la actividad sobre el archivo va a requerir mucho tiempo, será mejor que otras partes del código realicen otras acciones mientras se lee el mismo. Cuando se completa la lectura, la clase puede emitir un evento que indique "la lectura se ha completado". También se puede informar a otras partes del código cuando se emita este evento y el código puede realizar la acción indicada cuando reciba el evento de la clase que lo disparó. El tema de eventos se tratará posteriormente en su uso más común que son los de interfaz gráfica.

Los indexadores

Algunas de las clases pueden actuar como contenedores de otros valores. Estas son diseñadas para que otro código acceda a dichos valores ordenadamente como si fuera un vector. Por ejemplo, si se esta escribiendo una clase llamada DiasSinIndice que permite al código cliente

acceder a los valores de cadenas que nombran los días de la semana en orden y se desea que los elementos que realizan llamadas puedan obtener los valores de las cadenas, de modo que mediante algunos métodos públicos se permita a los elementos que realizan llamadas tener acceso a los valores. Una posible solución sería la siguiente.

Ejemplo

C#

```
namespace indexadores
{
    class DiasSinIndice
    {
        private string[] dias = { "Domingo", "Lunes", "Martes", "Miércoles",
              "Jueves", "Viernes", "Sábado"};
        public int ObtenerNumeroDeDias()
        {
            return 7;
        }
        public string ObtenerDia(int indiceDia)
        {
            if (indiceDia > ObtenerNumeroDeDias()) return "No existe";
            else return dias[indiceDia];
        }
    }
}
```

VB

```
Public Class DiasSinIndice
    Private dias() As String = {"Domingo", "Lunes", "Martes", "Miércoles", "Jueves",
"Viernes", "Sábado"}

    Public Function ObtenerNumeroDeDias() As Integer
        Return 7
    End Function

    Public Function ObtenerDia(indiceDia As Integer) As String
        If (indiceDia > ObtenerNumeroDeDias()) Then
            Return "No existe"
        Else
            Return dias(indiceDia)
        End If
    End Function
End Class
```


Un código cliente que quiera acceder a estos servicios sería como el siguiente.

Ejemplo

C#

```
        int cantidadDeDias;
        int indiceDeDia;
        DiasSinIndice diasDeLaSemanaSinIndexador = new DiasSinIndice();
```

```
        cantidadDeDias = diasDeLaSemanaSinIndexador.ObtenerNumeroDeDias();
        for (indiceDeDia = 0; indiceDeDia < cantidadDeDias; indiceDeDia++)
        {
                System.Console.WriteLine(
                        diasDeLaSemanaSinIndexador.ObtenerDia(indiceDeDia));
        }
```

VB

```
         Dim cantidadDeDias As Integer
         Dim indiceDeDia As Integer
         Dim diasDeLaSemanaSinIndexador As New DiasSinIndice()
         cantidadDeDias = diasDeLaSemanaSinIndexador.ObtenerNumeroDeDias()
         For indiceDeDia = 0 To cantidadDeDias - 1
System.Console.WriteLine(diasDeLaSemanaSinIndexador.ObtenerDia(indiceDeDia))
         Next
```

Si bien esto es correcto a nivel de código, también es cierto que el objeto se accede como un vector y sería deseable que el código lo reflejara como tal. Los indexadores permiten que se acceda a las clases como si fuera un vector. Para especificar el valor que se debe retornar cuando el código que realiza la llamada usa corchetes para acceder a un determinado valor que se encuentra almacenado dentro de la clase, se usa un fragmento de código llamado descriptor de acceso del indexador.

Teniendo esto en cuenta, se puede rescribir la clase DiasSinIndice como la clase DiasIndexados para que permita a los elementos que la llaman acceder a los nombres de los días usando corchetes en su sintaxis. Notar como en el nuevo código se eliminan los métodos y se remplazan por propiedades para acceder a los valores y el uso de la palabra clave `this` o `Me` (C# o VB) para indicar que es el objeto actualmente en ejecución el que retorna el valor en base a un índice.

Ejemplo

C#

```
namespace indexadores
{
    class DiasIndexados
    {
        private string[] dias = { "Domingo", "Lunes", "Martes", "Miércoles",
                                  "Jueves", "Viernes", "Sábado" };
        public int ObtenerNumeroDeDias
        {
            get
            {
                return 7;
            }
        }

        public string this[int indice]
        {
```

```
            get
            {
                if(indice > 7) return "No existe";
                return dias[indice];
            }
        }
    }
}
```

VB

```
Public Class DiasIndexados
    Private _dias() As String = {"Domingo", "Lunes", "Martes", "Miércoles",
"Jueves", "Viernes", "Sábado"}

    Public ReadOnly Property ObtenerNumeroDeDias() As Integer
        Get
            Return 7
        End Get
    End Property

    Default Public ReadOnly Property Dias(indice As Integer) As String
        Get
            If (indice > 7) Then
                Return "No existe"
            Else
                Return _dias(indice)
            End If
        End Get
    End Property


End Class
```

Nota: los indexadores en VB se manejan diferentes a C# ya que no existe una declaración específica para ellos en el lenguaje. La forma de declararlo es utilizando propiedades por defecto que son las que se llaman cuando no se especifica nada en el acceso que quiera lograr en el objeto

Un código cliente que quiera acceder a estos servicios sería como el siguiente.

Ejemplo

C#

```
int indiceDelDia;
DiasIndexados diasIndexados = new DiasIndexados();
string nombreDelDia;
for (indiceDelDia=0;indiceDelDia<diasIndexados.ObtenerNumeroDeDias; indiceDelDia++)
{
       nombreDelDia = diasIndexados[indiceDelDia];
       System.Console.WriteLine(nombreDelDia);
```

```
}
```

VB

```
        Dim indiceDelDia As Integer
        Dim _diasIndexados As New DiasIndexados()
        Dim nombreDelDia As String
        For indiceDelDia = 0 To _diasIndexados.ObtenerNumeroDeDias - 1
            nombreDelDia = _diasIndexados(indiceDelDia)
            System.Console.WriteLine(nombreDelDia)
        Next
        System.Console.WriteLine("Usando la propiedad por defecto directamente")
        System.Console.WriteLine(_diasIndexados.Dias(0))
        Console.ReadKey()
```

Los operadores

Un operador permite definir el comportamiento de la clase cuando se usa un objeto de su tipo en una expresión con un operador unario o binario. Esto significa que se puede ampliar el comportamiento de operadores predefinidos para que se ajusten a las necesidades de las clases. Por lo tanto, cuando se cree un objeto del tipo de la clase y se utilice el operador que se haya redefinido el comportamiento para objetos del tipo de la clase, se buscará en esta la redefinición (un método escrito en un formato especial que será explicado en el capítulo de conceptos avanzados de clases) y se realizará la operación en base a ella.

Otros tipos

Las clases pueden incluir en su definición otros tipos, como por ejemplo estructuras, enumeraciones o clases anidadas. Estas últimas serán explicadas en detalle en el capítulo de conceptos avanzados de clases.

Visibilidades: control de acceso a la clase

Uno de los beneficios de las clases es que pueden proteger sus variables y métodos miembros frente al acceso de otros objetos. ¿Por qué es esto importante? Bien, consideremos esto: se ha escrito una clase que representa una petición a una base de datos que contiene toda tipo de información secreta, es decir, registros de empleados o proyectos secretos de la compañía. Si se quiere mantener el control de acceso a esta información, sería bueno limitarlo.

Otra forma de ver lo mismo es para aquellos atributos que tiene un objeto que no deben cambiar. Por ejemplo, si existe una clase Persona, no debería existir forma en la clase o en un objeto que se cree a partir de ella, de cambiar el atributo nombre

Los datos que se almacenan dentro de las clases deben tener un acceso controlado ya que son la base de procesamiento de servicios y delimitan el estado de un objeto en un determinado momento del tiempo. Por lo tanto, su acceso debe ser controlado por algún mecanismo capaz de proteger sus valores y permita asegurar el correcto funcionamiento. Dicha herramienta existe en los lenguajes y se implementa a través de la visibilidad

Tanto en C# como en VB se puede utilizar los modificadores de acceso para proteger las variables, los métodos y cualquier elemento de la clase cuando se declaran. El lenguaje soporta cuatro niveles de acceso para las variables y métodos miembros: `public`, `private` (valor por defecto), `internal` y `protected` o `Public`, `Pivate` (valor por defecto), `Friend`, `Protected` (C# o VB)

La siguiente tabla muestra los posibles modificadores de acceso o visibilidad y como son afectados los elementos de una clase que son declarados con su uso

Modificador de Acceso	Significado
`public`	El acceso no está restringido.
`protected`	El acceso está limitado a la clase donde se realiza la declaración o tipos derivados de dicha clase.
`internal`	El acceso está limitado al proyecto actual.
`protected internal`	El acceso está limitado al proyecto actual o a los tipos derivados de la clase donde se realiza la declaración.
`private`	El acceso está limitado a la clase donde se realiza la declaración.

Sólo un modificador de acceso es permitido para un elemento declarado en una clase, excepto cuando se utiliza la combinación `protected internal` o `Protected Friend` (C# o VB). Los modificadores de acceso no están permitidos en los espacios de nombres. Los espacios de nombres no tienen restricciones de acceso.

Dependiendo del contexto en el que una declaración de miembro tiene lugar, sólo ciertas accesibilidades declaradas están permitidas. Si no hay ningún modificador de acceso que se especifique en la declaración de un miembro, se utiliza la accesibilidad por defecto que es privada. Por el momento se explicarán con mayor detalle las visibilidades `public`, `private` e `internal` o `Public`, `Private` y `Friend` (C# o VB). Posteriormente se ampliarán el resto de los modificadores.

`private` o `Private` (C# o VB)

El nivel de acceso **por defecto** y el más restringido es `private` o `Private` (C# o VB). Un miembro privado es accesible sólo para la clase en la que está definido. Se utiliza este acceso para declarar miembros que sólo deben ser utilizados por la clase. Esto incluye las variables que contienen información que si se accede a ella desde el exterior podría colocar al objeto en un estado de inconsistencia, o los métodos que llamados desde el exterior pueden poner en peligro el estado del objeto o del programa donde se está ejecutando. Para declarar un miembro privado se utiliza la palabra clave `private` o `Private` (C# o VB) en su declaración. La clase siguiente contiene una variable miembro y un método privados.

Ejemplo

C#

```
namespace declaraciones
{
    class Alfa
    {
        private int soyPrivado;
        private void metodoPrivado()
        {
            Console.WriteLine("metodoPrivado");
        }
    }
}
```

VB

```
Public Class Alfa
    Private soyPrivado As Integer
    Private Sub metodoPrivado()
        Console.WriteLine("metodoPrivado")
    End Sub
End Class
```

Los objetos del tipo Alfa pueden inspeccionar y modificar la variable soyPrivado y pueden invocar el método metodoPrivado(), pero los objetos de otros tipos no pueden acceder.

Ejemplo

C#

```
namespace declaraciones
{
    class Beta
    {
        void metodoAccesor()
        {
            Alfa a = new Alfa();
            a.soyPrivado = 10; // ilegal
            a.metodoPrivado(); // ilegal
        }
    }
}
```

VB

```
Public Class Beta
    Public Sub metodoAccesor()
        Dim a As Alfa = New Alfa()
        a.soyPrivado = 10 ' ilegal
        a.metodoPrivado() ' ilegal
    End Sub
End Class
```

La clase Beta definida aquí no puede acceder a la variable soyPrivado ni al método metodoPrivado() de un objeto del tipo Alfa porque tienen visibilidad `private` o `Private` (C# o VB) y Beta es una clase diferente a Alfa.

Si una clase está intentando acceder a una variable miembro a la que no tiene acceso, el compilador mostrará un mensaje de error similar a este y no compilará su programa:

```
1      'declaraciones.Alfa.soyPrivado' is inaccessible due to its protection level
```

Y si un programa intenta acceder a un método al que no tiene acceso, generará un error de compilación parecido a este:

```
2      'declaraciones.Alfa.metodoPrivado()' is inaccessible due to its protection level
```

`public` o `Public` (C# o VB)

El modificador de acceso más sencillo. Todas las clases, en todos los espacios de nombres tienen acceso a los miembros públicos de la clase siempre y cuando la clase sea declarada también con el modificador `public` o `Public` (C# o VB). Cuando una clase no se declara específicamente como `public` o `Public` (C# o VB) el valor por defecto será `internal` o `Friend` (C# o VB) y por lo tanto podrá crease un objeto **sólo desde el mismo ensamblado**.

Los miembros públicos se declaran sólo si su acceso no produce resultados indeseados al código externo a la clase que los utiliza. Por lo tanto, es la visibilidad más fácil de utilizar pero la más difícil de diseñar correctamente. Para declarar un miembro público se utiliza la palabra clave `public` o `Public` (C# o VB).

Ejemplo

C#

```
namespace griego
{
    public class Alfa
    {
        public int soyPublico;

        public void metodoPublico()
        {
            Console.WriteLine("metodoPublico");
        }
    }
}
```

VB

```
Public Class Alfa
    Public soyPublico As Integer
    Friend soyInterno As Integer

    Public Sub metodoPublico()
        Console.WriteLine("metodoPublico")
    End Sub
```

```
    Public Sub metodoInterno()
        Console.WriteLine("metodointerno")
    End Sub
End Class
```

Si se vuelve a escribir la clase Beta una vez más y se la coloca en un espacio de nombres distinto en un proyecto diferente que la clase Alfa, asegurándose que no están relacionadas (no es una subclase) de Alfa.

Ejemplo

C#

```
using griego;

namespace romano
{
    class Beta
    {
        public void metodoAccesor()
        {
            Alfa a = new Alfa();
            a.soyPublico = 10; // legal
            a.metodoPublico(); // legal
        }
    }
}
```

VB

```
Imports griego

Public Class Beta
    Public Sub metodoAccesor()
        Dim a As New Alfa()
        a.soyPublico = 10 ' legal
        a.metodoPublico() ' legal
    End Sub
End Class
```

Como se puede ver en el ejemplo anterior, Beta puede inspeccionar y modificar legalmente la variable soyPublico en la clase Alfa y puede llamar legalmente al método metodoPublico().

internal

La palabra clave interna es un modificador de acceso para tipos (clases) y miembros de tipos. Las clases o elementos miembro declarados explícitamente con el modificador de acceso `internal` o `Friend` (C# o VB) sólo son accesibles dentro de los archivos en el mismo ensamblado. Sin embargo, la omisión de la declaración es diferente para las clases que para los elementos miembro, como por ejemplo métodos y atributos. En el caso de las clases el valor por defecto es efectivamente `internal` o `Friend` (C# o VB) mientras que en los elementos de las clases el valor por defecto es `private`.

Para entender el concepto, se declara una clase llamada Beta en el espacio de nombres griego que posee un solo elemento con un modificador de acceso `internal` o `Friend` (C# o VB). El acceso por defecto de la clase es también `internal` o `Friend` (C# o VB) por lo antes mencionado. En otro proyecto, que se convertirá en otro ensamblado, se declara una referencia al proyecto griego, el cual por defecto creó un espacio de nombres con el mismo nombre del proyecto y se intentará acceder a la clase Beta desde el proyecto actual, llamado romano, en una clase llamada Gamma. Como en el proyecto romano existe una clase Beta, notar como se resuelve el acceso al espacio de nombres con notación de punto.

Ejemplo

C#

```
namespace romano
{
    class Gamma
    {
        public void metodoPublico()
        {
            // Error: Beta no es visible para el
            // espacio de nombres griego por estar
            // en otro ensamblado y ser internal
            // por más que tenga una referencia al
            // ensamblado
            // griego.Beta b = new griego.Beta();

            griego.Alfa a = new griego.Alfa();

            //a.metodoProtegido(); // ilegal
            //a.metodoInterno(); // ilegal
        }
    }
}
```

VB

```
Public Class Gamma
    Public Sub metodoPublico()
        ' Error: Beta no es visible para el
        ' espacio de nombres griego por estar
        ' en otro ensamblado y ser Friend
        ' Dim b As New griego.Beta()

        Dim a As New griego.Alfa()

        'a.metodoProtegido() ' ilegal
        'a.metodoInterno() ' ilegal
    End Sub
End Class
```

El error que arroja es el siguiente:

```
1     'griego.Beta.Beta()' is inaccessible due to its protection level
```

La razón del mismo es que la clase Beta se declaró con visibilidad `internal` ya que esta es su declaración por defecto al omitir el modificador de acceso y toda clase o elemento declarado de esta manera sólo es accesible desde el mismo ensamblado.

Nota: los modificadores `internal` y `protected` se retomarán luego de explicar herencia para su mejor comprensión

Espacios de nombres

Los espacios de nombres son grupos relacionados de clases, delegados, enumeraciones, estructuras e interfaces y proporcionan un mecanismo conveniente para manejar un gran juego de dichos elementos de los lenguajes y evitar los conflictos de nombres (porque los espacios de nombres en si mismos definen visibilidades). Para crearlos se utiliza la sentencia `namespace` o `Namespace` (C# ó VB).

Los espacios de nombres pueden anidarse, es decir, esta permitido declarar uno dentro del otro o en su defecto, realizar una declaración incluyendo un punto para separar los nombres desde el más externo al más interno. Cada elemento que se declare dentro de uno de ellos determina la ubicación lógica que dicho elemento tendrá y especificará la forma de accederlo cuando se resuelva la visibilidad de acceso al mismo.

Si se desea utilizar una clase que está dentro de un espacio de nombres diferente al actual cuando se escribe código (la situación más común es cuando desde un proyecto se referencia a otro y se quiere utilizar una clase de este último), se debe utilizar la palabra clave `using` ó `Imports` (C# ó VB) en el código para tener visibilidad y acceso. Estas declaraciones se realizan al comienzo de todo código fuente. Es común ver su utilización cuando se acceden a clases propias de .Net que se encuentran en sus respectivos espacios de nombres.

La sintaxis de la declaración de un espacio de nombres o el uso de otro en el código de una clase que se encuentre fuera del que se está declarando es:

C#

[< `using` "nombre del espacio de nombres[.subespacio de nombres[,…]]"]
    < declaración de la clase>+
[<`namespace` "nombre del espacio de nombres[.subespacio de nombres[,…]]">]

VB

[<`Imports` "nombre del espacio de nombres[.subespacio de nombres[,…]]"]
    < declaración de la clase>+
[<`Namespace` "nombre del espacio de nombres[.subespacio de nombres[,…]]">]

Ejemplo

C#

```
using System;

namespace clinica.utilidades
{
    class Horario
    {
        private int dia = 0;
        private int horaComienzo = 0;
        private int minutosComienzo = 0;
        private int horaFin = 0;
        private int minutosFin = 0;
        private int turnosPorHora = 0;


        public Horario(int dia, int horaComienzo, int minutosComienzo, int horaFin,
                int minutosFin, int turnosPorHora)
        {
            this.dia = dia;
            this.horaComienzo = horaComienzo;
            this.minutosComienzo = minutosComienzo;
            this.horaFin = horaFin;
            this.minutosFin = minutosFin;
            this.turnosPorHora = turnosPorHora;
        }

        public Horario(Horario f)
        {
            this.dia = f.dia;
            this.horaComienzo = f.horaComienzo;
            this.minutosComienzo = f.minutosComienzo;
            this.horaFin = f.horaFin;
            this.minutosFin = f.minutosFin;
        }

        public Horario agregar(int masDias)
        {
            Horario nuevaFecha = new Horario(this);
            nuevaFecha.dia += nuevaFecha.dia;
            return nuevaFecha;
        }

        public void imprimir()
        {
            Console.WriteLine("Horario: ");
            Console.WriteLine("Día: " + dia);
            Console.WriteLine("Hora de comienzo: " + horaComienzo);
            Console.WriteLine("Minutos de comienzo: " + minutosComienzo);
            Console.WriteLine("Hora de fin: " + horaFin);
            Console.WriteLine("Minutos de fin: " + minutosFin);
            Console.WriteLine("Turnos por hora:" + turnosPorHora);
        }
```

```
        public int Dia
        {
            get
            {
                return dia;
            }
        }
        public int HoraComienzo
        {
            get
            {
                return horaComienzo;
            }
        }
        public int MinutosComienzo
        {
            get
            {
                return minutosComienzo;
            }
        }
        public int HoraFin
        {
            get
            {
                return horaFin;
            }
        }
        public int MinutosFin
        {
            get
            {
                return minutosFin;
            }
        }
        public int TurnosPorHora
        {
            get
            {
                return turnosPorHora;
            }
        }
    }
}

using System;
using System.Collections.Generic;
using clinica.utilidades;

namespace clinica
{
    namespace medicos
    {
        class Medico
        {
```

```
private String nombre;
private String apellido;
private String especialidad;
private int turnosPorHora = 0;
private HashSet<Horario> horarios = new HashSet<Horario>();

public String Nombre
{
    get
    {
        return nombre;
    }
    set
    {
        nombre = value;
    }
}
public String Apellido
{
    get
    {
        return apellido;
    }
    set
    {
        apellido = value;
    }
}
public String Especialidad
{
    get
    {
        return especialidad;
    }
    set
    {
        especialidad = value;
    }
}
public int TurnosPorHora
{
    get
    {
        return turnosPorHora;
    }
    set
    {
        turnosPorHora = value;
    }
}
public HashSet<Horario> Horarios
{
    get
    {
        return horarios;
```

```
                }
                set
                {
                    horarios = value;
                }
            }
        }
    }
}
```

VB

```
Public Class Horario
    Private _dia As Integer = 0
    Private _horaComienzo As Integer = 0
    Private _minutosComienzo As Integer = 0
    Private _horaFin As Integer = 0
    Private _minutosFin As Integer = 0
    Private _turnosPorHora As Integer = 0

    Public Sub New(ByVal _dia As Integer, ByVal _horaComienzo As Integer, _
                ByVal _minutosComienzo As Integer, _
                ByVal _horaFin As Integer, ByVal _minutosFin As Integer, _
                ByVal turnosPorHora As Integer)
        Me._dia = _dia
        Me._horaComienzo = _horaComienzo
        Me._minutosComienzo = _minutosComienzo
        Me._horaFin = _horaFin
        Me._minutosFin = _minutosFin
        Me._turnosPorHora = _turnosPorHora
    End Sub

    Public Sub New(ByVal f As Horario)
        Me._dia = f._dia
        Me._horaComienzo = f._horaComienzo
        Me._minutosComienzo = f._minutosComienzo
        Me._horaFin = f._horaFin
        Me._minutosFin = f._minutosFin
        Me._turnosPorHora = f._turnosPorHora
    End Sub

    Public Function agregar(ByVal masDias As Integer) As Horario
        Dim nuevaFecha As New Horario(Me)
        nuevaFecha._dia += masDias
        Return nuevaFecha
    End Function

    Public Sub imprimir()
        Console.WriteLine("Horario: ")
        Console.WriteLine("Día: " + _dia.ToString)
        Console.WriteLine("Hora de comienzo: " + _horaComienzo.ToString)
        Console.WriteLine("Minutos de comienzo: " + _minutosComienzo.ToString)
        Console.WriteLine("Hora de fin: " + _horaFin.ToString)
        Console.WriteLine("Minutos de fin: " + _minutosFin.ToString)
        Console.WriteLine("Turnos por hora:" + _turnosPorHora.ToString)
    End Sub
```

```
    Public Property Dia() As Integer
        Get
            Return _dia
        End Get
        Set(ByVal value As Integer)
            _dia = value
        End Set
    End Property
    Public Property HoraComienzo() As Integer
        Get
            Return _horaComienzo
        End Get
        Set(ByVal value As Integer)
            _horaComienzo = value
        End Set
    End Property
    Public Property MinutosComienzo() As Integer
        Get
            Return _minutosComienzo
        End Get
        Set(ByVal value As Integer)
            _minutosComienzo = value
        End Set
    End Property
    Public Property HoraFin() As Integer
        Get
            Return _horaFin
        End Get
        Set(ByVal value As Integer)
            _horaFin = value
        End Set
    End Property
    Public Property MinutosFin() As Integer
        Get
            Return _minutosFin
        End Get
        Set(ByVal value As Integer)
            _minutosFin = value
        End Set
    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

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

```
    Private _horarios As New HashSet(Of Horario)

    Public Property Nombre() As String
        Get
            Return _nombre
        End Get
        Set(ByVal value As String)
            _nombre = value
        End Set
    End Property
    Public Property Apellido() As String
        Get
            Return _apellido
        End Get
        Set(ByVal value As String)
            _apellido = value
        End Set
    End Property
    Public Property Especialidad() As String
        Get
            Return _especialidad
        End Get
        Set(ByVal value As String)
            _especialidad = value
        End Set
    End Property
    Public Property TurnosPorHora() As Integer
        Get
            Return _turnosPorHora
        End Get
        Set(ByVal value As Integer)
            _turnosPorHora = value
        End Set
    End Property
    Public Property Horarios() As HashSet(Of Horario)
        Get
            Return _horarios
        End Get
        Set(ByVal value As HashSet(Of Horario))
            _horarios = value
        End Set
    End Property

End Class
```


Nota: La clase `HashSet` se explicará posteriormente en el capítulo de colecciones

Si se está implementando un grupo de clases que representan una serie de objetos gráficos como círculos, rectángulos, líneas o puntos y se quiere que estas clases estén disponibles para otros programadores, se las puede poner en un mismo espacio de nombres, por ejemplo, `graficos` y entregar el ensamblado a los programadores (junto con alguna documentación de referencia

como, por ejemplo, qué hacen las clases y los interfaces además de cuáles interfaces de las clases –métodos– son públicas).

De esta forma, otros programadores pueden determinar fácilmente para qué es el grupo de clases, cómo utilizarlas, y cómo relacionarlas unas con otras o, también, con otras clases y espacios de nombres. Los nombres de clases no tienen conflictos con los nombres de las clases de otros espacios de nombres porque dichas clases e interfaces dentro de un espacio de nombres son referenciados en términos del mismo (técnicamente espacio de nombres crea una visibilidad nueva que le pertenece).

Si se diera por algún motivo una ambigüedad entre dos clases que tengan el mismo nombre pero estén es espacios de nombres diferentes, esta se puede resolver usando el nombre totalmente calificado de la clase, lo cual significa colocar junto al nombre de la clase todos los espacios y subespacios de nombres que determinan la ubicación lógica de la misma separando cada uno con puntos.

Ejemplo

C#

```
namespace graficos
{
    class Rectangulo
    {
    }
}

namespace graficos
{
    class Circulo
    {
    }
}
```

VB

```
Public Class Rectangulo

End Class

Public Class Circulo

End Class
```

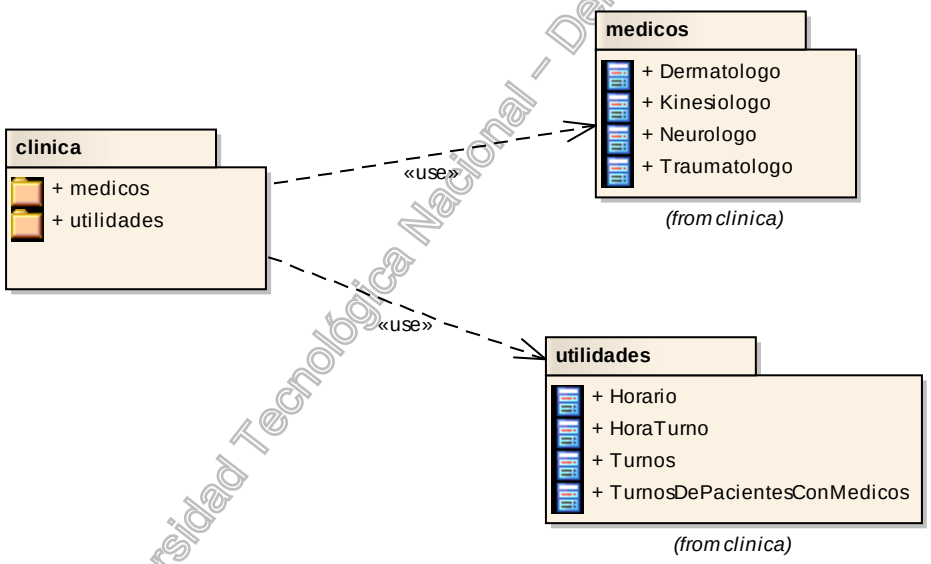
Nota: VB define el espacio de nombres por defecto con el mismo nombre del proyecto. Para ver su declaración, remitirse a las propiedades del proyecto

La primera línea del código anterior crea un espacio de nombres llamado `graficos`. Todas las clases definidas en un archivo que contiene esta sentencia son miembros del espacio de nombres. Por lo tanto, `Circulo`, y `Rectangulo` son miembros del espacio de nombres `graficos`.

Los espacios de nombres ayudan a manejar grandes sistemas de software porque agrupan clases en su interior que en conjunto pueden representar una funcionalidad específica de un sistema. Por ejemplo, pueden contener clases, subclases e interfaces que definen dicha funcionalidad.

Como se mencionó anteriormente, los espacios de nombres definen visibilidades para acceder a ellos. Por eso, el último nivel de acceso es el que se obtiene si no se especifica ningún otro nivel de acceso a los miembros (subespacios de nombres). No se accederán a otros espacios de nombres aunque sean subespacios del actual si no se especifican explícitamente. Este nivel de acceso permite que las clases tengan acceso a los miembros de las clases definidas en él.

Un ejemplo gráfico en UML del manejo de un espacio de nombres, (por creación y uso) se puede ver en la siguiente figura



Otro ejemplo puede ser, una nueva versión de la clase Alfa en la que se declara una variable y un método con acceso de ensamblado. Alfa reside en el espacio de nombres griego.

Ejemplo

C#

```
namespace griego
{
    public class Alfa
    {
```

```
        public int soyPublico;
        internal int soyInterno;

        public void metodoPublico()
        {
            Console.WriteLine("metodoPublico");
        }

        public void metodoInterno()
        {
            Console.WriteLine("metodointerno");
        }
    }
}
```

VB

```
Public Class Alfa
    Public soyPublico As Integer
    Friend soyInterno As Integer

    Public Sub metodoPublico()
        Console.WriteLine("metodoPublico")
    End Sub

    Public Sub metodoInterno()
        Console.WriteLine("metodointerno")
    End Sub
End Class
```

La clase Alfa tiene acceso a sus elementos internos como soyInterno y a metodoInterno().

Además, todas las clases declaradas dentro del mismo ensamblado que Alfa también tienen acceso a soyInterno y metodoInterno(). Si Alfa y Beta están declaradas como parte del ensamblado con espacio de nombres griego podría Beta acceder a los elementos internos de Alfa.

Ejemplo

C#

```
namespace griego
{
    class Beta
    {
        internal void metodoAccesor()
        {
            Alfa a = new Alfa();
            a.soyPublico = 10; // legal
            a.metodoPublico(); // legal
            a.soyInterno = 20; // legal
            a.metodoInterno(); // legal
        }
    }
}
```

VB

```
Friend Class Beta
    Friend Sub metodoAccesor()
        Dim a As Alfa = New Alfa()
        a.soyPublico = 10 ' legal
        a.metodoPublico() ' legal
        a.soyInterno = 20 ' legal
        a.metodoInterno() ' legal
    End Sub
End Class
```

Entonces Beta puede acceder legalmente a soyInterno y a metodoInterno().

Para utilizar una clase específica o cualquier otro elemento declarado en un espacio de nombres en el archivo actual se utiliza la sentencia `using` ó `Imports` (C# ó VB). Esta debe estar al principio del archivo antes que cualquier definición.

Esto provoca que la clase y/o el interfaz estén disponibles para su uso por las clases, interfaces, delegados, enumeraciones, eventos o estructuras definidos en el archivo que se esté elaborando.

Si intenta utilizar una clase o cualquier otro elemento desde un espacio de nombres al que no se ha se ha invocado con la sentencia `using` ó `Imports` (C# ó VB), el compilador mostrará un error.

Nota: Es probable que el mismo nombre para una clase se encuentre en más de un espacio de nombres. En estos casos se debe colocar el nombre totalmente calificado en al menos una de ellas (la que no se encuentre mediante la sentencia `using` ó `Imports` - C# ó VB) para diferenciarlas. El nombre totalmente calificado implica colocar la secuencia de nombres desde la raíz hasta aquel en que se encuentre la clase en cuestión.

Usando espacios de nombres

Los espacios de nombres son muy utilizados en programas de dos maneras. En primer lugar, las clases del Framework de .NET utilizan espacios de nombres para organizar sus muchas clases. En segundo lugar, declarar espacios de nombres propios puede ayudar a controlar el ámbito de nombres de clase y métodos en proyectos de programación más grandes.

La mayoría de los archivos de código fuente comienzan con una sección de directivas `using` ó `Imports` (C# ó VB). En esta sección se enumeran los espacios de nombres que la aplicación va a utilizar con frecuencia, y evita que el programador especifique un nombre completo cada vez que se utiliza un método de una clase que lo contiene.

Por ejemplo, mediante la inclusión de la línea:

C#

```
using System;
```

VB

```
Using System
```

Al comienzo de un archivo de código fuente, el programador puede utilizar el código:

C#

```
Console.WriteLine();
```

VB

```
Console.WriteLine()
```

En lugar de

C#

```
System.Console.WriteLine();
```

VB

```
System.Console.WriteLine()
```

La directiva `using` ó `Imports` (C# ó VB) también se puede utilizar para crear un alias para un espacio de nombres. Por ejemplo, si utiliza un espacio de nombres previamente escrito que contenga espacios de nombres anidados, es posible que desee declarar un alias para proporcionar una forma abreviada de referirse a alguno en particular, así:

C#

```
using miAlias = System.Console;
```

VB

```
Imports miAlias = System.Console
```

Permite utilizar una declaración del siguiente tipo:

C#

```
miAlias.WriteLine();
```

VB

```
miAlias.WriteLine()
```

Usando nombres totalmente calificados

Los espacios de nombres y tipos tienen títulos únicos descritos por nombres totalmente calificados que indican una jerarquía lógica. Por ejemplo, la declaración A.B implica que A es el nombre del espacio de nombres o tipo, y B está anidado dentro de él.

En el siguiente ejemplo, hay clases y espacios de nombres anidados. El nombre totalmente calificado es indicado como un comentario después de cada entidad.

Ejemplo

C#

```
namespace N1     // N1
{
    public class C1      // N1.C1
    {
        public class C2   // N1.C1.C2
        {
        }
    }
    namespace N2  // N1.N2
    {
        public class C2   // N1.N2.C2
        {
        }
    }
}

namespace N1.N2
{
    public class C3   // N1.N2.C3
    {
    }
}
```

VB

```
Public Class C1      ' N1.C1

    Public Class C2 ' N1.C1.C2

    End Class

End Class
Namespace N2  ' N1.N2
    Public Class C2   ' N1.N2.C2

    End Class
End Namespace

Namespace N2
    Public Class C3

    End Class
End Namespace
```


El siguiente programa demuestra la utilización de los espacios de nombre realizando las declaraciones de objetos utilizando los nombres totalmente calificados. Notar que los modificadores de visibilidad de los tipos superiores (clases) y del tipo anidado C2 son los que permiten el acceso para la creación de los objetos una vez resuelto el espacio de nombres.

Ejemplo

C#

```
namespace calificados
{
    class Program
    {
        static void Main(string[] args)
        {
            N1.C1 c1 = new N1.C1();
            N1.C1.C2 c2= new N1.C1.C2();
            N1.N2.C3 c3 = new N1.N2.C3();
        }
    }
}
```

VB

```
odule Module1

    Sub Main()
        Dim c1 As New N1.C1()
        Dim c2 As N1.C1.C2 = New N1.C1.C2()
        Dim c3 As N1.N2.C3 = New N1.N2.C3()
    End Sub

End Module
```

Por supuesto el uso de los nombres totalmente calificados se hace necesario cuando se necesita resolver visibilidades. En un ejemplo como el anterior, se puede calificar el espacio de nombre en la sentencia `using` ó `Imports` (C# ó VB) sin necesidad de hacerlo en cada declaración. Sin embargo, cuando se presentan ambigüedades, la forma de resolución es utilizando el nombre totalmente calificado cada vez que se deba resolver la visibilidad.

Ejemplo

C#

```
using System;

namespace espaciosDeNombres
{
    namespace libros
    {
        namespace inventario
        {
            class Stock
            {
                public void AgregarInventario()
                {
                 Console.WriteLine("Agregando inventario via AgregarInventario()!");
                }
            }
        }
```

```
    }
}
```

VB

```
Namespace espaciosDeNombres
    Namespace libros
        Namespace inventario
            Public Class Stock
                Public Sub AgregarInventario()
                    Console.WriteLine( _
                            "Agregando inventario via AgregarInventario()!")
                End Sub
            End Class
        End Namespace
    End Namespace
End Namespace
```

El programa que quieras acceder a la clase Stock deberá resolver la visibilidad para el acceso. En este caso, se utiliza `using` ó `Imports` (C# ó VB) para lograrlo.

Ejemplo

C#

```
using System;
using EspaciosDeNombresCS.Libros.Inventario;

namespace espacios
{
    class Program
    {
        static void Main(string[] args)
        {
            Stock stk = new Stock();
            stk.AgregarInventario();
            Console.ReadKey();
        }
    }
}
```

VB

```
Imports espacios.espaciosDeNombres.libros.inventario

Module Module1
    Sub Main()
        Dim stk As New Stock()
        stk.AgregarInventario()
        Console.ReadKey()
    End Sub
End Module
```

El espacio de nombres global

En Microsoft. Net, cada programa se crea a partir de un espacio de nombres predeterminado. Este espacio de nombres por defecto se llama el espacio de nombres global. Pero el programa en sí puede declarar cualquier número de espacios de nombres, cada uno con un nombre único. La ventaja es que cada espacio de nombres puede contener cualquier número de clases, interfaces, estructuras, etc y también otros espacios de nombres, cuyos nombres sean únicos sólo dentro de dicho espacio de nombres. Los miembros con el mismo nombre se pueden crear en algún otro espacio de nombres sin ningún error que arroje el compilador de Microsoft. Net. Sin embargo, al momento de acceder, se debe resolver adecuadamente la visibilidad para lograr acceso.

La posibilidad de acceder a un miembro en el espacio de nombres global es útil cuando el miembro puede ser ocultado por otra entidad del mismo nombre. Se debe tener en cuenta que en cuestiones de visibilidades siempre gana la que se sea menos externo (conocido como resolución en base a la localidad). Esto quiere decir que si se encuentra un nombre más interno en la jerarquía de declaraciones que otro, este **ocultará** la declaración más externa y la misma se deberá resolver para lograr acceso.

Por ejemplo, en el código siguiente, Console se resuelve por defecto como ResolucionVisibilidad.Console en lugar de con el tipo Console en el espacio de nombres System.

Ejemplo

C#

```
namespace espacioGlobal
{
    class ResolucionVisibilidad
    {
        // Definir una clase nueva llamada 'System' para causar problemas.
        // y ocultar el espacio de nombres System
        public class System { }

        // Definir una constante llamada 'Console' para causar más problemas.
        // y ocultar la clase Console
        const int Console = 7;
        const int numero = 66;

        static void Main()
        {
            // Error: Accede a ResolucionVisibilidad.Console
            // Console.WriteLine(number);
        }
    }
}
```

VB

```
Public Class ResolucionVisibilidad
    ' Definir una clase nueva llamada 'System' para causar problemas.
    ' y ocultar el espacio de nombres System
```

```
    Public Class System

    End Class

    ' Definir una constante llamada 'Console' para causar más problemas.
    ' y ocultar la clase Console
    Public Const Console As Integer = 7
    Public Const numero As Integer = 66
    Shared Sub Main()
        ' Error: Accede a ResolucionVisibilidad.Console
        ' Console.WriteLine(number)
        ' Error: Accede a ResolucionVisibilidad.System
        ' System.Console.WriteLine(number)
    End Sub
End Class
```

Usando System.Console todavía resulta en un error porque el espacio de nombres System está oculto por la clase ResolucionVisibilidad.System

Ejemplo

C#

```
            // Error: Accede a ResolucionVisibilidad.System
             System.Console.WriteLine(number);
```

VB

```
        ' Error: Accede a ResolucionVisibilidad.System
        System.Console.WriteLine(number)
```

Sin embargo, puede evitar este error mediante el uso de `global::System.Console.WriteLine` o `Global::System.Console.WriteLine` (C# o VB). Cuando el identificador de la izquierda es `global` o `Global` (C# o VB) la búsqueda para el identificador de la derecha se inicia en el espacio de nombres global. Por ejemplo, la siguiente declaración hace referencia a ResolucionVisibilidad como miembro del espacio `global` o `Global` (C# o VB).

Ejemplo

C#

```
namespace espacioGlobal
{
    class ResolucionVisibilidad
    {
        // Definir una clase nueva llamada 'System' para causar problemas.
        // y ocultar el espacio de nombres System
        public class System { }

        // Definir una constante llamada 'Console' para causar más problemas.
        // y ocultar la clase Console
        const int Console = 7;
        const int numero = 66;
```

```
        static void Main()
        {
            // Error: Accede a ResolucionVisibilidad.Console
            // Console.WriteLine(number);
            // Error: Accede a ResolucionVisibilidad.System
            // System.Console.WriteLine(number);
            // Esto está bien!!!
            global::System.Console.WriteLine(numero);
            global::System.Console.ReadKey();
        }
    }
}
```

VB

```
Public Class ResolucionVisibilidad
    ' Definir una clase nueva llamada 'System' para causar problemas.
    ' y ocultar el espacio de nombres System
    Public Class System

    End Class

    ' Definir una constante llamada 'Console' para causar más problemas.
    ' y ocultar la clase Console
    Public Const Console As Integer = 7
    Public Const numero As Integer = 66
    Shared Sub Main()
        ' Error: Accede a ResolucionVisibilidad.Console
        ' Console.WriteLine(number)
        ' Error: Accede a ResolucionVisibilidad.System
        ' System.Console.WriteLine(number)
        ' Esto está bien!!!
        Global.System.Console.WriteLine(numero)
        Global.System.Console.ReadKey()
    End Sub
End Class
```


Obviamente, la creación de espacios de nombres propios denominados System no es recomendable, y es poco probable que se encuentre esto en cualquier código. Sin embargo, en grandes proyectos, es una posibilidad muy real que la duplicación de espacio de nombres se pueda producir de una forma u otra. En estas situaciones, el calificador de espacio de nombres `global` o `Global` (C# o VB) es garantía porque se puede especificar el espacio de nombres raíz.

Sobrecarga de Métodos

.Net soporta la sobrecarga de métodos, por eso varios métodos pueden compartir el mismo nombre. Por ejemplo, si se ha escrito una clase que puede proporcionar varios tipos de datos (cadenas, enteros, etc...) en un área de dibujo, se podría escribir un método que supiera como tratar a cada tipo de dato. En otros lenguajes no orientados a objetos, se tendría que pensar un nombre distinto para cada uno de los métodos. dibujaCadena(), dibujaEntero, etc...

Se puede utilizar el mismo nombre para todos los métodos pasándole un tipo de parámetro o cantidad diferente en cada uno de los métodos. Entonces en la clase de dibujo, se podrán declarar tres métodos llamados dibujar() y que cada uno aceptara un tipo de parámetro diferente.

Ejemplo

C#

```
namespace sobrecarga
{
    class DibujodeDatos
    {
        public void Dibujar(String s)
        {
            Console.WriteLine("Dibujando con un string: " + s);
        }
        public void Dibujar(int i)
        {
            Console.WriteLine("Dibujando con un int: " + i);
        }
        public void Dibujar(float f)
        {
            Console.WriteLine("Dibujando con un float: " + f);
        }
    }
}
```

VB

```
Public Class DibujodeDatos
    Public Sub Dibujar(ByVal s As String)
        Console.WriteLine("Dibujando con un string: " + s)
    End Sub
    Public Sub Dibujar(ByVal i As Integer)
        Console.WriteLine("Dibujando con un entero: " + i.ToString)
    End Sub
    Public Sub Dibujar(ByVal si As Single)
        Console.WriteLine("Dibujando con un string: " + si.ToString)
    End Sub
End Class
```


Los métodos son diferenciados por el compilador basándose en el número y tipo de sus argumentos. Para ello, redefine internamente los nombres de los métodos valiéndose de las declaraciones de parámetros. De esta manera, los métodos internamente serían:

```
DibujarString
Dibujarint
Dibujarfloat
```

Por lo tanto internamente son todos distintos. Generalizando la forma de uso, suponiendo las siguientes declaraciones en C# (es análogo para VB):

```
        public void Metodo(float f, int i);
        public void Metodo(float f);
        public void Metodo(String s);
```

Internamente serían:

```
       metodofloatint
       metodofloat
       metodoString
```

Motivo por el cual se deben poner los nombres con diferente cantidad o tipo de argumentos.

Nota: declarar un parámetro como `ref` u `out` o `ByRef` (C# o VB) no diferencia a los mismos para la sobrecarga.

Nota: Los valores devueltos por un método no los diferencian en la sobrecarga. Por lo tanto dos métodos con distinto valor retornado pero el mismo tipo de argumentos, genera un error. Por otra parte, si los argumentos son diferentes, no importa si el valor retornado es igual o distinto.

Constructores sobrecargados

Los constructores, al igual que cualquier método, se pueden sobrecarga. Las reglas a seguir por éstos son las mismas que para los métodos, con la salvedad que un constructor sólo pertenece a la clase en la que se lo declara y por lo tanto las reglas se aplican sólo en la clase, mientras que la única condición de sobrecarga de métodos en este aspecto es que sean visibles entre si, con lo cual los métodos pueden ser sobrecargados a lo largo de una cadena de herencia.

Ejemplo

C#

```
namespace constructores.defecto
{
    class Ejemplo
    {
        private int atrib1;
        private char atrib2;
        public Ejemplo(int a) { atrib1 = a; }
        public Ejemplo() { atrib1 = 5; }
        public void metodo()
        {
            //[sentencias;]
        }
    }
}

namespace constructores.sobrecargados
{
    class Ejemplo
    {
```

```
        private int atrib1;
        private char atrib2;
        public Ejemplo(int a) { atrib1 = a; }
        public Ejemplo(char a) { atrib2 = a; }
        public void metodo()
        {
            //[sentencias;]
        }
    }
}

namespace constructores
{
    class Program
    {
        static void Main(string[] args)
        {
            Ejemplo e = new Ejemplo(8);
            constructores.defecto.Ejemplo e2 = new constructores.defecto.Ejemplo(9);
            constructores.sobrecargados.Ejemplo e3 =
                    new constructores.sobrecargados.Ejemplo(10);
            constructores.sobrecargados.Ejemplo e4 =
                    new constructores.sobrecargados.Ejemplo('a');
        }
    }
}
```

VB

```
Namespace defecto
    Public Class Ejemplo
        Private atrib1 As Integer
        Private atrib2 As Char

        Public Sub New(ByVal a As Integer)
            atrib1 = a
        End Sub

        Public Sub New()
            atrib1 = 5
        End Sub

        Public Sub metodo()
            '[sentencias]
        End Sub
    End Class
End Namespace

Namespace sobrecargados
    Public Class Ejemplo
        Private atrib1 As Integer
        Private atrib2 As Char

        Public Sub New(ByVal a As Integer)
            atrib1 = a
        End Sub
```

```
        Public Sub New(ByVal a As Char)
            atrib2 = a
        End Sub

        Public Sub metodo()
            '[sentencias]
        End Sub
    End Class
End Namespace

Module Module1

    Sub Main()
        Dim e1 As Ejemplo = New Ejemplo(8)
        Dim e2 As constructores.defecto.Ejemplo = _
              New constructores.defecto.Ejemplo(9)
        Dim e3 As constructores.sobrecargados.Ejemplo = _
              New constructores.sobrecargados.Ejemplo(10)
        Dim e4 As constructores.sobrecargados.Ejemplo = _
              New constructores.sobrecargados.Ejemplo("a"c)
    End Sub

End Module
```


¿Cómo el Common Language Runtime realiza la recolección de basura?

Las computadoras no tienen una cantidad infinita de memoria, por lo que la memoria debe ser recuperada cuando una variable o un objeto ya no la requiere. Los tipos por valor se destruyen y su memoria es recuperada cuando salen del ámbito (alcance, como el bloque de sentencias o cuerpo de un método) en el que se declaran. Este proceso es manejado por el código que el compilador genera al crear una aplicación, y ocurre en una forma muy determinista. En contraste, el recolector de basura (garbage collector), que forma parte del CLR, reclama la memoria que usan los tipos por referencia. Este comportamiento es mucho menos determinista y no es controlado directamente por el código.

Los procesos de creación y destrucción de un objeto

Se crea un objeto mediante el operador `new` (C#) o `New` (VB). Los ejemplos de código siguiente crean una nueva instancia de la clase TextBox.

Desde el punto de vista del programador, la operación `new` o `New` es atómica, pero en el fondo, la creación de objetos es en realidad un proceso de dos fases:

1. La operación `new` o `New` asigna un bloque de memoria en bruto del heap. No se tiene control sobre esta fase de la creación de un objeto.
2. La operación `new` o `New` convierte el bloque de memoria en bruto en un objeto y tiene que inicializar el objeto. Se puede controlar esta fase mediante un constructor.

Después de haber creado un objeto, se puede acceder a sus miembros, y se pueden crear otras variables de referencia que se refieran al mismo objeto, como se muestra en los ejemplos de código siguientes.

Ejemplo

C#

```
TextBox mensaje = new TextBox(); // TextBox es un tipo por referencia
```

VB

```
Dim mensaje As New TextBox() ' TextBox es un tipo por referencia
```

Se pueden crear tantas referencias al mismo objeto como la aplicación lo requiera. Esto tiene un impacto en la vida de un objeto. El CLR tiene que hacer un seguimiento de todas estas referencias. Si la variable mensaje desaparece al salir del ámbito (visibilidad), otras variables que se refieren al mismo objeto, como, por ejemplo, refMensaje, aún pueden existir. La vida útil de un objeto puede no estar atada a una variable de referencia en particular. El CLR sólo puede destruir un objeto y recuperar su memoria ocupada cuando **todas las referencias a él han desaparecido**.

Al igual que la creación de objetos, la destrucción de objetos es también un proceso de dos fases. Las dos fases de la destrucción de objetos exactamente un espejo de las dos fases de la creación de estos:

1. El CLR, opcionalmente, realiza algún tipo de procesamiento a medida para poner en orden y liberar recursos que utiliza el objeto. Se puede controlar esta fase escribiendo un destructor.
2. El CLR desasigna la memoria que utiliza el objeto y lo devuelve al heap. No se tiene control sobre esta fase.

El CLR realiza estos dos pasos utilizando el recolector de basura.

Escribiendo destructores

Se puede utilizar un destructor para realizar cualquier limpieza que se requiera cuando un objeto es recogido como basura. El CLR llama al destructor de un objeto después de la última referencia a él ha desaparecido. La sintaxis para escribir un destructor varía significativamente entre C # y VB: en C #, se define un método con el mismo nombre que la clase, pero precedido del símbolo tilde (~), y en Visual Basic, se define un `Protected Overrides Sub` que se denomina Finalize.

Ejemplo

C#

```
namespace destructores
{
    class Clase1
    {
        public Clase1()
```

```
        {
            Console.WriteLine("Construyendo la clase 1");
        }
        ~Clase1()
        {
            Console.WriteLine("Destruyendo la clase 1");
        }
    }
}

namespace destructores
{
    class Clase2
    {
        public Clase2()
        {
            Console.WriteLine("Construyendo la clase 2");
        }
        ~Clase2()
        {
            Console.WriteLine("Destruyendo la clase 2");
        }
    }
}

namespace destructores
{
    class Clase3
    {
        public Clase3()
        {
            Console.WriteLine("Construyendo la clase 3");
        }
        ~Clase3()
        {
            Console.WriteLine("Destruyendo la clase 3");
            Console.ReadKey();
        }
    }
}

namespace destructores
{
    class Program
    {
        static void Main(string[] args)
        {
            Clase1 c1 = new Clase1();
            c1 = null;

            Clase2 c2 = new Clase2();
            c2 = null;

            Clase3 c3 = new Clase3();
            c3 = null;
```

```
            Console.WriteLine("Notar que al destructor lo llama el garbage" +
                              " collector");

            Console.WriteLine("");
            Console.WriteLine("Presione un tecla para salir.");
            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class Clase1
    Public Sub New()
        Console.WriteLine("Construyendo la clase 1")
    End Sub

    Protected Overrides Sub Finalize()
        Console.WriteLine("Destruyendo la clase 1")
        MyBase.Finalize()
    End Sub
End Class

Public Class Clase2
    Public Sub New()
        Console.WriteLine("Construyendo la clase 2")
    End Sub

    Protected Overrides Sub Finalize()
        Console.WriteLine("Destruyendo la clase 2")
        MyBase.Finalize()
    End Sub
End Class

Public Class Clase3
    Public Sub New()
        Console.WriteLine("Construyendo la clase 3")
    End Sub

    Protected Overrides Sub Finalize()
        Console.WriteLine("Destruyendo la clase 3")
        MyBase.Finalize()
    End Sub
End Class

Module Module1
    Sub Main()
        Dim c1 As New Clase1()
        c1 = Nothing

        Dim c2 As New Clase2()
        c2 = Nothing

        Dim c3 As New Clase3()
        c3 = Nothing
```

```
        Console.WriteLine("Notar que al destructor lo llama el garbage collector")

        Console.WriteLine("")
        Console.WriteLine("Presione un tecla para salir.")
        Console.ReadKey()
    End Sub
End Module
```

Nota: en el código del ejemplo anterior la destrucción comienza luego de presionar una tecla y puede ser muy rápido de manera que no se vean todos los carteles sacados por consola

Hay algunas muy restricciones importantes que se aplican a los destructores:

- Los destructores sólo se aplican a los tipos por referencia. No se puede declarar un destructor en un tipo por valor, como una estructura o enumeración.
- Si se está utilizando C#, no se puede especificar un modificador de acceso a un destructor. Si se está utilizando VB, se debe especificar que el método Finalize es `Protected`. Nunca se llama al destructor desde el código propio. Sólo el recolector de basura invoca al destructor.
- El destructor no puede tener ningún parámetro. Una vez más, esto se debe a que nunca se llama al destructor de forma explícita.

Se debe tener en cuenta que el compilador de C# traduce automáticamente un destructor en una rescritura del método Object.Finalize. Por ejemplo, el compilador convierte el destructor de la clase de Contador en el código que es similar a la que se muestra a continuación.

Ejemplo

C#

```
protected override void Finalize()
{
    try
    {
      …
     }
    finally { base.Finalize(); }
}
```

El método Finalize que el compilador de C# genera contiene el cuerpo del destructor dentro de un bloque `try`, seguido de un bloque `finally` que llama al método Finalize de la clase base (los bloques `try-catch-finally` o `try-catch-finally` (C# o VB) se explicarán posteriormente en el capítulo de excepciones y sirven para el manejo de situaciones anómalas como condiciones de error).

Esto asegura que un destructor siempre llama al destructor de la clase base. Es importante darse cuenta de que sólo el compilador puede hacer esta traducción. No se puede remplazar el método Finalize rescribiéndolo, y no se lo puede llamar de forma explícita en el código.

¿Por qué utilizar el recolector de basura?

Nunca se puede destruir un objeto por sí mismo usando código de C # o VB. Hay buenas razones por las que los diseñadores del Framework .NET decidieron prohibir hacerlo. Si se delega al programador la responsabilidad de destruir los objetos, tarde o temprano, una de las siguientes situaciones se presentan:

- Olvidarse de destruir un objeto. Esto significaría que el destructor del objeto no se pueda ejecutar, la limpieza no ocurriría y la memoria no se reasignaría de nuevo al heap. Se podría fácilmente quedar sin memoria.
- Se puede tratar de destruir a un objeto activo. Recordar que los objetos se acceden por la referencia. Si un objeto contiene una referencia a otro objeto que ya se ha destruido, esta referencia puede terminar apuntando tanto a memoria no utilizada como, posiblemente, a un objeto completamente diferente que pase a ser asignado al mismo lugar de la memoria. En cualquier caso, el resultado de la utilización de esta referencia sería indefinido en el mejor de los casos o un riesgo para la seguridad en el peor.
- Se puede tratar de destruir el mismo objeto más de una vez. Esto podría o no ser desastroso, según el código en el destructor.

Estos problemas son inaceptables en el código que está diseñado para ser ejecutado de manera verificable por el CLR, el cual sitúa en su lista de objetivos de diseño la robustez y la alta seguridad.

En cambio, el recolector de basura se hace responsable de la destrucción de objetos en lugar del usuario. El recolector de basura brinda las siguientes garantías:

- Cada objeto será destruido y se ejecutará su destructor. Cuando el programa termina, todos los objetos pendientes serán destruidos.
- Todos los objetos serán destruidos exactamente una vez.
- Todos los objetos serán destruidos sólo cuando se conviertan en inalcanzables, es decir, cuando no hay referencias que apunten al objeto.

Estas garantías son tremendamente útiles y libradoras de tediosas tareas de limpieza en las que son fáciles las equivocaciones. Estas permiten concentrarse en la lógica propia del programa y ser más productivo.

¿Cuándo se produce la recolección de basura? Esto puede parecer una pregunta extraña. Después de todo, seguramente la recolección de basura se produce cuando un objeto ya no es necesario. Bueno, sí, pero no necesariamente de inmediato. La recolección de basura puede ser un proceso costoso, por lo que en tiempo de ejecución se recoge la basura sólo cuando es necesario (cuando

se piensa que la memoria disponible está empezando a agotarse), y luego recoge todo lo que pueda.

Nota: Se puede invocar al recolector de basura en un programa mediante una llamada al método estático System.GC.Collect. Sin embargo, salvo en algunos pocos casos, esto no es recomendable. El método System.GC.Collect inicia el recolector de basura, pero el proceso se ejecuta de forma asincrónica y, cuando la llamada al método se completa, todavía no se sabe si los objetos han sido destruidos. Se debe dejar que el CLR decida cuándo es mejor recoger la basura.

Una característica del recolector de basura es que no se sabe y no debe basarse en el orden en que los objetos se destruyen. El último punto a entender es sin duda el más importante: los destructores no se ejecutan hasta que los objetos son recogidos como basura. Si se escribe un destructor, se sabe que va a ser ejecutado, pero no cuándo.

¿Cómo funciona el recolector de basura?

El recolector de basura se ejecuta en su propio hilo (thread) y puede ejecutarse sólo en ciertos momentos, por lo general, cuando la aplicación llegue al final de un método y el CLR decide que sería beneficioso recuperar memoria. Mientras se ejecuta, otros threads que se estén ejecutando en la aplicación temporalmente son detenidos. Esto se debe a que el recolector de basura puede ser que necesite mover objetos a su alrededor para acomodarlos y actualizar las referencias a objetos y no puede hacer esto mientras que los objetos están en uso. Los pasos que el recolector de basura toma son los siguientes:

1. Construye un mapa de todos los objetos accesibles. Hace esto siguiendo reiteradamente campos de referencia dentro de los objetos. El recolector de basura se basa este mapa con mucho cuidado y se asegura que referencias circulares no lo hagan caer en una recursión infinita. Cualquier objeto que no está en este mapa se considera inalcanzable.
2. Comprueba si alguno de los objetos inalcanzables tiene un destructor que se debe ejecutar (un proceso que se llama la finalización). Cualquier objeto inalcanzable que requiere la finalización se coloca en una cola especial que se llama la cola freachable (finalización accesible).
3. Borra de la memoria los objetos inalcanzables restantes (los que no requieren de finalización) moviendo los objetos accesibles hacia abajo en el heap. Esta acción desfragmenta el heap y libera la memoria en la parte superior del mismo. Cuando el recolector de basura mueve un objeto accesible, también actualiza las referencias al mismo.
4. En este punto, que permite que otros threads continúen.
5. Finaliza los objetos inalcanzables que requieren finalización (aquellos que se encuentran en la cola freachable) utilizando su propio hilo.

Recomendaciones

Escribir clases que contienen destructores añade complejidad al código, al proceso de recolección de basura y se hace que el programa se ejecute más lentamente. Si el programa no contiene destructores, el recolector de basura no necesita colocar objetos inalcanzables en la cola freachable y finalizarlos. Utilizar los destructores sólo cuando realmente se los necesita.

Se debe tener mucho cuidado cuando se escribe un destructor. En particular, tener en cuenta que, si el destructor llama a los métodos de otros objetos, el recolector de basura puede haber limpiado ya los otros objetos. Recordar que el orden de finalización no está garantizado.

Por lo tanto, asegurarse que los destructores no dependan unos de otros o se superpongan entre sí (por ejemplo, no tener dos destructores que traten de liberar el mismo recurso).

Considerar la implementación de la interfaz IDisposable y la creación de un objeto con la declaración `using` (C#) o `Using` (VB) para que ayude a asegurarse que el destructor para el objeto se ejecuta en una forma más determinista. Esta instrucción permite, entre otras cosas, definir una visibilidad ya que se le puede asociar un bloque y paréntesis en los cuales realizar declaraciones. Cuando dicho bloque termina, todas las declaraciones y recursos tomados se liberan. Este tema se explicará con más detalle en el capítulo de conceptos avanzados de clases.

Nota: La interfaz IDisposable se describe más adelante en el tema "Implementando la interfaz IDisposable" en este módulo.

Uso de `this` y `Me`

La palabra reservada `this` o `Me` (C# o VB) posee la referencia al objeto actualmente en ejecución y se utiliza para encontrar los elementos que pertenecen a dicho objeto. Como se mencionó anteriormente, este valor se almacena en el stack para cada método en ejecución y es por esta referencia que se encuentran las variables de instancia de un objeto.

Sólo los elementos de un objeto, métodos, atributos, etc…, poseen asociada una referencia del tipo `this` o `Me` (C# o VB), por lo tanto se lo puede utilizar para resolver ambigüedades.

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a los atributos del mismo. Sin embargo, algunas veces no se querrá tener ambigüedad sobre el nombre de la variable de instancia y uno de los argumentos del método que tengan el mismo nombre.

Por ejemplo, si una clase tiene declarados atributos que tienen el mismo nombre que los argumentos de un método, utilizando `this` o `Me` (C# o VB) se resuelve la ambigüedad.

Ejemplo

C#

```
namespace ambiguedad
{
    class Ejemplo
```

```
    {
        private int atrib1;
        private char atrib2;
        public Ejemplo(int a) { atrib1 = a; }
        public Ejemplo(char a) { atrib2 = a; }
        public Ejemplo(char atrib2, int atrib1)
        {
            this.atrib1 = atrib1;
            this.atrib2 = atrib2;
        }
        public void metodo()
        {
            //[sentencias;]
            atrib2 = 'a';
        }
    }
}
```

VB

```
Public Class Ejemplo
    Private atrib1 As Integer
    Private atrib2 As Char
    Public Sub New(ByVal a As Integer)
        atrib1 = a
    End Sub

    Public Sub New(ByVal a As Char)
        atrib2 = a
    End Sub

    Public Sub New(ByVal atrib2 As Char, ByVal atrib1 As Integer)
        Me.atrib1 = atrib1
        Me.atrib2 = atrib2
    End Sub

    Public Sub metodo()
        '[sentencias]
        atrib2 = "a"c
    End Sub
End Class
```


Otro punto importante a tener en cuenta es que todo elemento que pertenece a un objeto tiene asociado una referencia `this` o `Me` (C# o VB) porque esta es la forma de encontrar los miembros que pertenecen a cada objeto en particular. Pero se debe ser cuidadoso al evaluar esta afirmación. Por ejemplo, un método de un objeto utiliza `this` o `Me` (C# o VB) para encontrar las variables de instancia y por lo tanto siempre tiene un `this` o `Me` (C# o VB) asociado. Sin embargo, los parámetros y variables locales del mismo no tienen asociado un `this` o `Me` (C# o VB) ya que se alojan en el stack. Si un método no tiene un `this` o `Me` (C# o VB) asociado, no pertenece al objeto por más que este declarado dentro de la clase que lo define. Este es el caso de los métodos estáticos como se verá posteriormente.

Uso de `this` en Constructores (C#)

Se puede utilizar `this` en la línea de declaración de un constructor luego del operador de dos puntos, para invocar otro constructor de la misma clase.

Como se mencionó anteriormente, `this` almacena la referencia al objeto actualmente en ejecución. Si se está construyendo el objeto, la suma de esta palabra reservada con el operador de llamado a función, C# lo resuelve invocando al constructor que posee los argumentos pasados como parámetros dentro el operador de llamado a función utilizado con `this`

Ejemplo

C#

```
namespace constructorConThis
{
    class Ejemplo
    {
        public Ejemplo(int a) { atrib1 = a; }
        public Ejemplo(int a, float b):this(a)
        {
            atrib2 = b;
        }
        private int atrib1;
        private float atrib2;
        public void metodo()
        {
            //[sentencias;]
        }
    }
}
```

Por lo visto hasta el momento, se puede aprovechar la sobrecarga y las características de `this` en el llamado a constructores de la propia clase para utilizarlos en conjunto.

Ejemplo

C#

```
namespace constructorConThis
{
    class Empleado
    {
        private String nombre;
        private Fecha fechaNacimiento;
        private double salario;
        private const double SALARIO_BASE = 15000.00;

        // Constructor
        public Empleado(String nombre, Fecha fDN, double salario)
        {
            this.nombre = nombre;
            this.fechaNacimiento = fDN;
            this.salario = salario;
        }
```

```
        public Empleado(String nombre, double salario)
            : this(nombre, null, salario)
        {

        }

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

        }

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

        }
    }
}
```

Uso de `Me.New` en Constructores (VB)

Se puede utilizar `Me.New` en la primera línea de un constructor para invocar otro constructor de la misma clase.
Como se mencionó anteriormente, `Me` almacena la referencia al objeto actualmente en ejecución. Si se está construyendo el objeto, la suma de esta palabra reservada con New, VB lo resuelve invocando al constructor que posee los argumentos pasados como parámetros dentro el operador de llamado a función utilizado con `Me.New`. El llamado debe ser la primera instrucción dentro del constructor de invocación.

Ejemplo

VB

```
Public Class Ejemplo
    Private atrib1 As Integer
    Private atrib2 As Single

    Public Sub New(ByVal a As Integer)
        atrib1 = a
    End Sub

    Public Sub New(ByVal a As Integer, ByVal b As Single)
        Me.New(a)
        atrib2 = b
    End Sub

    Public Sub metodo()
        '[sentencias]
    End Sub
End Class
```

Por lo visto hasta el momento, se puede aprovechar la sobrecarga y las características de `Me.New` en el llamado a constructores de la propia clase para utilizarlos en conjunto.

Ejemplo

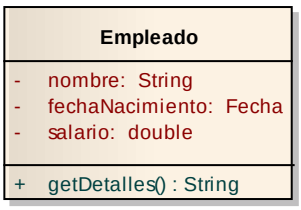
VB

```
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

    ' más código de Empleado...
End Class
```

Herencia

Para comenzar a desarrollar este tema, primero se plantea la necesidad del mismo desde el punto de vista del código. Por lo tanto se supone una clase simple con atributos y métodos como se describe a continuación en el diagrama UML y el código que se deriva de él.

Ejemplo



C#

```
namespace herencia
{
    class Empleado
    {
        private String nombre;
        private double salario = 1500.00;
        private Fecha fechaNacimiento;

        public 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;
            }
        }
        public Fecha FechaNacimiento
        {
            get
            {
                return fechaNacimiento;
            }
            set
            {
                fechaNacimiento = value;
            }
        }
    }
}
```

VB

```
Public Class Empleado
    Private Dim _nombre As String
    Private Dim _salario As Double = 1500.0
    Private Dim _fechaNacimiento As Fecha
```

```
    Public Function getDetalles() As String
        Return "Nombre: " + _nombre + "\nSalario: " + _salario
    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
    Public Property FechaNacimiento() As Fecha
        Get
            Return _fechaNacimiento
        End Get
        Set(ByVal value As Fecha)
            _fechaNacimiento = value
        End Set
    End Property
End Class
```

En .Net las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama subclase. La clase de la que está derivada se denomina superclase.

Las subclases heredan el estado y el comportamiento en forma de variables y métodos de su superclase. La subclase puede utilizar los ítems heredados de su superclase tal y como son, o puede modificarlos o rescribirlos. Por eso, según se recorre la cadena de la herencia, las clases se convierten en más y más especializadas.

> *Una subclase es una clase que desciende de otra clase. Una subclase hereda el manejo del estado y el comportamiento de su superclase.*

Declarar la Superclase de una clase

Se declara que una clase es una subclase de otra clase en la declaración de la misma clase. Por ejemplo, si se quiere crear una subclase llamada SubClase de otra clase llamada SuperClase, la forma de escribirlo es:

C#

```
class SubClase : SuperClase
{
. . .
```

```
}
```

VB

```
Public Class SubClase
    Inherits SuperClase
. . .
End Class
```

Esto declara que SubClase es una subclase de SuperClase. Y también declara implícitamente que SuperClase es la superclase de SubClase. Una subclase también hereda variables y miembros de la superclase de su superclase, y así a lo largo de la cadena de herencia. Para hacer esta explicación un poco más sencilla, *cuando se haga referencia a la superclase de una clase significa el ancestro más directo* (la clase que figura en la declaración luego de :).

> *Cuando se haga referencia a una clase que antecede a otra se la denomina superclase de la subcadena de herencia. Si la superclase en particular es la primera de la cadena de herencia, se la denomina superclase de la cadena de herencia.*

Para especificar explícitamente la superclase de una clase, se debe poner el operador : además del nombre de la superclase entre el nombre de la clase que se ha creado y la llave de comienzo del cuerpo de la clase, como se mencionó anteriormente y se muestra a continuación.

Ejemplo

C#

```
class Gerente : Empleado
{
. . .
}
```

VB

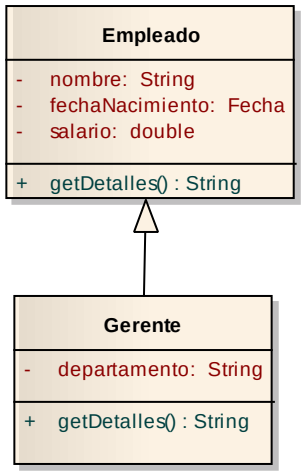
```
Public Class Gerente
    Inherits Empleado
. . .
End Class
```

Esto declara explícitamente que la clase Empleado es la superclase de Gerente. Declarar esto implica implícitamente que Gerente es una subclase de Empleado. Una subclase hereda las variables, métodos y otros elementos de su superclase.

Crear una subclase puede ser tan sencillo como incluir el operador : o `Inherits` (C# o VB) en su declaración de clase. Sin embargo, se tendrán que tener en cuenta otras posibilidades que se derivan de esto en su código cuando se crea una subclase, como rescribir métodos, lo cual se explicará posteriormente.

Basándose en los ejemplos anteriores, el código para las clases Empleado y Gerente se puede plantear como muestra el diagrama UML y el código que se deriva de él.

Ejemplo



Una clase en C# o VB sólo puede tener una superclase directa. No está soportada la herencia múltiple de clases concretas.

Existe un mecanismo que simula una herencia múltiple a través de declarar interfaces, pero este es un tema que se expondrá posteriormente

Se puede definir la sintaxis apropiada de la herencia, basado en la sintaxis UML, de la siguiente manera:

C#

```
<modificador> class <nombre> [: < superclase>]
{
                    <declaraciones>*

}
```

VB

```
<modificador> Class <nombre>
    [Inherits <superclase>]
      <declaraciones>*
End Class
```


Clases que no se pueden heredar

Toda clase declarada con el modificador de acceso `sealed` o `NotInheritable` (C# o VB) no puede ser heredada por una subclase. Este tipo de modificador de acceso previene de generar subclases de una superclase que lo defina.

Son clases que, por lo general a causa de una decisión de diseño, no deben poseer subtipos.

¿Qué variables miembro hereda una subclase?

*Una subclase hereda todas las variables de instancia de su superclase que puedan ser **visibles** desde la subclase (a menos que el atributo se oculte por una declaración).*

Esto es, las subclases:

- Heredan aquellas variables miembros declaradas como `public`, `protected`, `internal` y `protected internal` o `Public`, `Protected`, `Friend` y `Protected Friend` (C# o VB)
- La declaración `protected internal` o `Protected Friend` (C# o VB) permiten la herencia, pero el acceso con notación de punto sólo dentro del mismo ensamblado.
- No hereda las variables miembro `private` o `Private` (C# o VB)
- No se heredan aquellas variables miembros declaradas sin modificador de acceso ya que su definición por defecto es `private` o `Private` (C# o VB)
- No hereda las variables de instancia de la superclase si la subclase declara una variable que utiliza el mismo nombre. El atributo de la subclase oculta a la variable miembro de la superclase por defecto, pero se debe utilizar el operador apropiado para que la declaración sea correcta.

Ocultar Variables Miembro

Como se mencionó en la sección anterior, los atributos definidos en la subclase pueden ocultar las variables miembro o de instancia que tienen el mismo nombre en la superclase.

Así como esta característica es poderosa y conveniente, también puede ser una fuente de errores: ocultar una variable miembro puede hacerse deliberadamente o por accidente. Entonces, cuando se use nombres para las variables de instancia hay que ser cuidadoso y ocultar sólo aquellas que realmente se desean.

Una posibilidad interesante de los atributos es que una clase puede acceder a aquellos que se encuentren ocultos en su superclase gracias a que se puede resolver la visibilidad. En C# cuando se oculta una variable miembro la declaración correcta debería realizarse utilizando `new` **como modificador, no como operador**. En VB la palabra clave que se utiliza con el mismo fin es `Shadows`. Sin embargo se admite la omisión del mismo porque es el valor por defecto y el compilador sólo genera una advertencia.

Ejemplo

C#

```
new internal float unNumero = 3.3F;
```

VB

```
Friend Shadows unNumero As Single = 3.29999995F
```

Utilizando con el valor por defecto del modificador quedaría

Ejemplo

C#

```
internal float unNumero = 3.3F;
```

VB

```
Friend Shadows unNumero As Single = 3.29999995F
```

Por ejemplo, observar las declaraciones en esta pareja de superclase y subclase:

Ejemplo

C#

```
namespace herencia
{
    class Sub: Super
    {
        new internal float unNumero = 3.3F;

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

        public float RetornaValor()
        {
            return unNumero + base.unNumero;
        }
    }
}
```

VB

```
Public Class SubC
    Inherits Super
    Friend Shadows unNumero As Single = 3.29999995F
    Public Sub New()
        Dim ej As Integer = MyBase.unNumero
    End Sub

    Public Function RetornaValor() As Single
        Return unNumero + MyBase.unNumero
    End Function
End Class
```


La variable unNumero de Sub oculta a la variable unNumero de Super. Pero se puede acceder a la variable de la superclase utilizando:

C#

```
base.unNumero
```

VB

```
MyBase.unNumero
```

`base` o `MyBase` (C# o VB) es una palabra clave que permite a un método referirse a las variables ocultas y métodos sobrescritos de una superclase.

¿Qué métodos hereda una Subclase?

La regla que especifica los métodos heredados por una subclase es similar a la de las variables miembro.

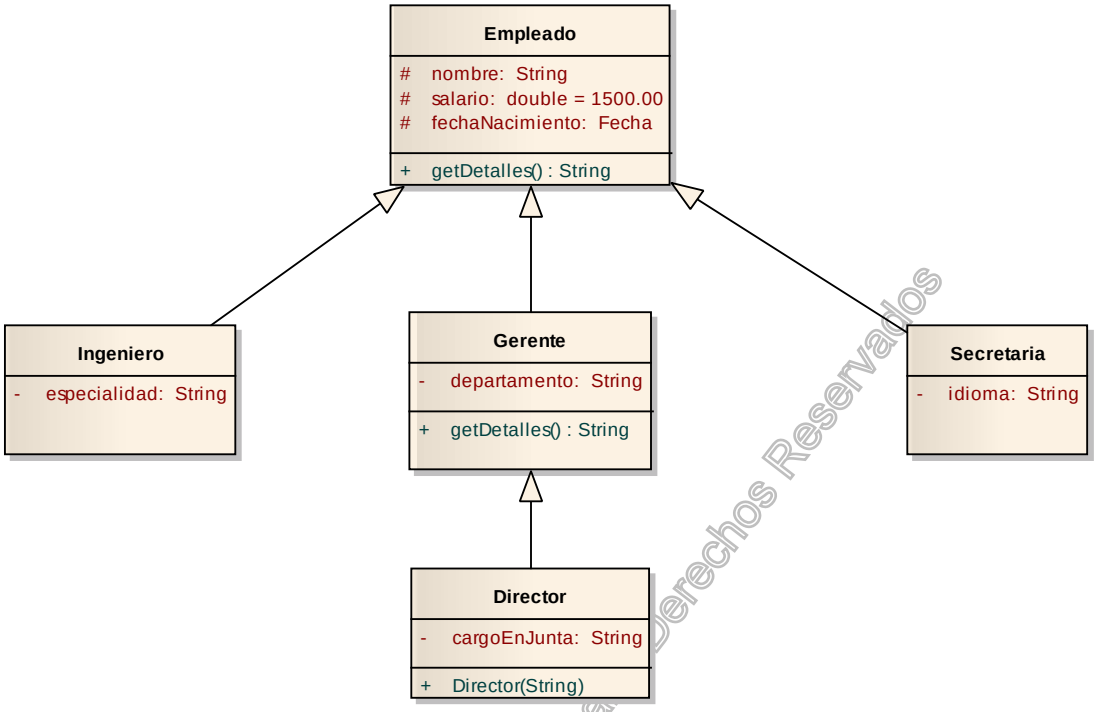
*Una subclase hereda todos los métodos de su superclase que son **visibles** para la subclase.*

Esto es, una Subclase:

- Hereda aquellos métodos declarados como `public`, `protected`, `internal` y `protected internal` o `Public`, `Protected`, `Friend` y `Protected Friend` (C# o VB)
- La declaración `protected internal` o `Protected Friend` (C# o VB) permiten la herencia, pero el acceso con notación de punto sólo dentro del mismo ensamblado.
- No hereda los métodos `private` o `Private` (C# o VB)
- No se heredan aquellos métodos declarados sin modificador de acceso ya que su definición por defecto es `private` o `Private` (C# o VB)
- No hereda un método de la superclase si la subclase declara un método que utiliza el mismo nombre. Se dice que el método de la subclase sobrescribe al método de la superclase.

En C# y VB, una clase puede heredar de tan sólo una superclase. Sin embargo, una superclase lo puede ser de muchas subclases y formar así distintas cadenas de herencia. Por ejemplo, en el siguiente gráfico UML se muestran tres cadenas de herencia que convergen a la misma superclase.

Ejemplo



El código que representa a este gráfico es el siguiente

C#

```
namespace herencia
{
    class Empleado
    {
        protected String nombre;
        protected double salario = 1500.00;
        protected Fecha fechaNacimiento;

        public String getDetalles()
        {
            return "Nombre: " + nombre + "\nSalario: " + salario;
        }
    }
}


namespace herencia
{
    class Gerente : Empleado
    {
        private String departamento;

        public String getDetalles()
```

```
        {
            return "detalles";
        }

        public String Departamento
        {
            get { return departamento; }
            set { departamento = value; }
        }
    }
}



namespace herencia
{
    class Director : Empleado
    {
        private String cargoEnJunta;

        public String CargoEnJunta
        {
            get { return cargoEnJunta; }
            set { cargoEnJunta = value; }
        }

    }
}



namespace herencia
{
    class Secretaria : Empleado
    {
        private String idioma;

        public String Idioma
        {
            get { return idioma; }
            set { idioma = value; }
        }
    }
}



namespace herencia
{
    class Ingeniero : Empleado
    {
        private String especialidad;

        public String Especialidad
```

```
        {
            get { return especialidad; }
            set { especialidad = value; }
        }
    }
}
```

VB

```
Public Class Empleado
    Private Dim _nombre As String
    Private Dim _salario As Double = 1500.0
    Private Dim _fechaNacimiento As Fecha

    Public Function getDetalles() As String
        Return "Nombre: " + _nombre + "\nSalario: " + _salario
    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
    Public Property FechaNacimiento() As Fecha
        Get
            Return _fechaNacimiento
        End Get
        Set(ByVal value As Fecha)
            _fechaNacimiento = value
        End Set
    End Property
End Class



Public Class Gerente
    Inherits Empleado
    Private _departamento As String

    Public Function getDetalles() As String
        Return "detalles"
    End Function
    Public Property Departamento() As String
        Get
            Return _departamento
        End Get
```

```
        Set(ByVal value As String)
            _departamento = value
        End Set
    End Property
End Class



Public Class Director
    Inherits Gerente
    Private cargoEnJunta As String
    Public Property CargoEnJunta1() As String
        Get
            Return cargoEnJunta
        End Get
        Set(ByVal value As String)
            cargoEnJunta = value
        End Set
    End Property
End Class



Public Class Secretaria
    Inherits Empleado
    Private _idioma As String
    Public Property Idioma() As String
        Get
            Return _idioma
        End Get
        Set(ByVal value As String)
            _idioma = value
        End Set
    End Property
End Class



Public Class Ingeniero
    Inherits Empleado
    Private _especialidad As String
    Public Property Especialidad() As String
        Get
            Return _especialidad
        End Get
        Set(ByVal value As String)
            _especialidad = value
        End Set
    End Property
End Class
```


Nota: El código en VB genera una advertencia porque no se declaran los métodos correctamente en las subclases. Esto es debido a que tienen el mismo nombre que en la superclase y VB exige declararlos de otra manera. La forma correcta se abordará más adelante cuando se explique

rescritura

El modificador `protected` y `Protected`

Permite a la propia clase y las subclases que accedan a los miembros declarados con este modificador. Este nivel de acceso se utiliza cuando es apropiado para una subclase de la clase tener acceso a los miembros.

Ejemplo

C#

```
namespace griego
{
    public class Alfa
    {
        public int soyPublico;
        protected int soyProtegido;
        internal int soyInterno;

        public void metodoPublico()
        {
            Console.WriteLine("Método Público");
        }

        protected void metodoProtegido()
        {
            Console.WriteLine("Método Protegido");
        }

        internal void metodoInterno()
        {
            Console.WriteLine("Método Interno");
        }
    }
}
```

VB

```
Public Class Alfa
    Public soyPublico As Integer
    Protected soyProtegido As Integer
    Friend soyInterno As Integer

    Public Sub metodoPublico()
        Console.WriteLine("metodoPublico")
    End Sub

    Protected Sub metodoProtegido()
        Console.WriteLine("Método Protegido")
    End Sub

    Friend Sub metodoInterno()
        Console.WriteLine("metodointerno")
    End Sub
```

```
End Class
```

Por otra parte, la clase Gama, también está declarada como miembro del paquete griego (y no es una subclase de Alfa). La clase Gama no puede acceder legalmente al miembro estoyProtegido del objeto Alfa y ni lo puede llamar legalmente.

Ejemplo

C#

```
namespace griego
{
    class Gama
    {
        public void metodoAccesor()
        {
            Alfa a = new Alfa();
            Beta b = new Beta();
            //a.estoyProtegido = 10; // ilegal
            //a.metodoProtegido(); // ilegal
            a.metodoInterno();
            b.metodoAccesor();
        }

    }
}
```

VB

```
Public Class Gamma
    Public Sub metodoAccesor()
        Dim a As New Alfa()
        Dim b As New Beta()
        'a.soyProtegido = 10 ' ilegal
        'a.metodoProtegido() ' ilegal
        a.metodoInterno()
        b.metodoAccesor()
    End Sub
End Class
```


Para ver como afecta el modificador `protected` a una subclase de Alfa, se introduce una nueva clase, Delta, que es subclase de Alfa pero reside en un paquete diferente, latin. La clase Delta puede acceder tanto a estoyProtegido como a metodoProtegido(), pero solo en objetos del tipo Delta o sus subclases. La clase Delta no puede acceder a estoyProtegido o metodoProtegido() en objetos del tipo Alfa.metodoAccesor() en el siguiente ejemplo intenta acceder a la variable miembro estoyProtegido de un objeto del tipo Alfa, que es ilegal, y en un objeto del tipo Delta que es legal. Similarmente, metodoAccesor() intenta invocar a metodoProtegido() en un objeto del tipo Alfa, que también es ilegal:

Ejemplo

C#

```
namespace latin
{
    class Delta : Alfa
    {
        void metodoAccesor()
        {
            Alfa a = new Alfa();
            a.estoyProtegido = 10; // ilegal
            soyProtegido = 10; // legal
            a.metodoProtegido(); // ilegal
            metodoProtegido(); // legal
        }

    }
}
```

VB

```
Imports griego

Public Class Delta
    Inherits Alfa
    Sub metodoAccesor()
        Dim a As New Alfa()
        a.soyProtegido = 10 ' ilegal
        soyProtegido = 10 ' legal
        a.metodoProtegido() ' ilegal
        metodoProtegido() ' legal
    End Sub
End Class
```

Cabe destacar que el llamado al método estoyProtegido es legal porque se invoca a través de la visibilidad que tiene Delta por heredar de Alfa. Si una clase es una subclase de la clase con el miembro protegido, la clase tiene acceso a éste por acceso directo como se explicó anteriormente en el caso de una subclase.

El modificador `protected internal` o `Protected Friend`

Cuando se quiere que los elementos de una clase provean acceso a los subtipos o a cualquier clase que se encuentre en el mismo ensamblado por medio de la notación de punto, se deben combinar las declaraciones `protected` e `internal` o `Protected` y `Friend` (C# o VB).

Si el elmento de la clase se encuentra en otro ensamblado, se debe heredar la clase para poder tener acceso a los elementos definidos como `protected internal` o `Protected Friend` (C# o VB), ya que no son visibles de otra manera.

Sobrescribir métodos

En muchas oportunidades, el comportamiento de una operación definida en la superclase no es adecuado y se debe modificar para el servicio que brindará la subclase. Una subclase puede sobrescribir completamente la implementación de un método heredado o puede mejorar el método añadiéndole funcionalidad y cambiar así el servicio que presta, de manera que sea adecuado a su implementación.

> *Cuando se sobrescribe un método en una subclase, se oculta la visibilidad del método que existe en la superclase.*

Remplazar la implementación de un método de una superclase

Algunas veces, una subclase puede necesitar remplazar completamente la implementación de un método de su superclase. De hecho, muchas superclases proporcionan implementaciones de métodos vacías con el propósito que la mayoría, si no todas, sus subclases remplacen completamente la implementación de ese método.

Para remplazar completamente la implementación de un método de la superclase, simplemente se llama a un método con el mismo nombre que el del método de la superclase y se sobrescribe el método con la misma firma (prototipo) que la del método sobrescrito, **incluyendo el valor a retornar definido** (distinto al caso de la sobrecarga).

Para rescribir correctamente un método se deben seguir cuatro reglas básicas:

1. El método de la subclase debe tener el mismo prototipo (firma de la función) del de la superclase
2. El modificador de acceso de la clase derivada no puede ser de menor acceso que el definido en la superclase
3. Si el método de la superclase es el primero que se va a definir para luego ser rescrito, debe definirse con la palabra clave `virtual` u `Overridable` (C# o VB)
4. El método de la subclase que rescribe un método virtual debe definirse con la palabra clave `override` u `Overrides` (C# o VB) y las sucesivas clases derivadas que sobrescriban el mismo método deben hacerlo también

Ejemplo

C#

```
namespace rescritura
{
    class MiClase
    {
        internal bool unaVariable;

        internal virtual void unMetodo()
        {
            unaVariable = true;
```

```
        }
    }
}
```

VB

```
Public Class MiClase
    Friend unaVariable As Boolean

    Friend Overridable Sub unMetodo()
        unaVariable = True
    End Sub
End Class
```

En una clase derivada que se desee rescribir el método se define lo siguiente

Ejemplo

C#

```
namespace rescritura
{
    class OtraClase : MiClase
    {
        new internal bool unaVariable;

        internal override void UnMetodo()
        {
            unaVariable = false;
            base.UnMetodo();
            Console.WriteLine(unaVariable);
            Console.WriteLine(base.unaVariable);
        }
    }
}
```

VB

```
Public Class OtraClase
    Inherits MiClase
    Friend Shadows unaVariable As Boolean

    Friend Overrides Sub unMetodo()
        unaVariable = False
        MyBase.unMetodo()
        Console.WriteLine(unaVariable)
        Console.WriteLine(MyBase.unaVariable)
    End Sub
End Class
```


De esta manera, el método UnMetodo de MiClase queda oculto para los objetos que se declaren del tipo OtraClase, es decir, cuando se cree un objeto del tipo OtraClase y se invoque al método UnMetodo por notación de punto (lo cual implica que debe ser accesible por este medio) **siempre se llamará al definido en OtraClase**. Si por algún motivo se quisiera invocar al método de la superclase, sólo se podrá hacer dentro del método definido en la subclase, OtraClase, resolviendo

visibilidad (si simplemente se llamara a UnMetodo, se crearía un llamado rentrante, es decir, se llamaría a sí mismo) se puede acceder y esto se logra mediante la palabra clave `base` o `MyBase` (C# o VB) que indica que el método a acceder es el de la clase base o superclase. Esta palabra clave sirve para resolver visibilidades respecto de la superclase con cualquier elemento y se puede ver como se utiliza en el método y el atributo para ganar acceso a los declarados en MiClase.

La palabra reservada `base` o `MyBase` (C# o VB) se usa para resolver visibilidad. Para entender la mecánica de esto, se debe comprender como se ocultan elementos de una superclase en una subclase, lo cual sucede cuando tienen la superclase o la subclase declarados exactamente el mismo identificador, ya sea un atributo o un método de la clase. Este hecho, como se explicó anteriormente se denomina rescritura y el resultado de ella es ocultar los miembros de la superclase.

Si una declaración oculta alguna variable miembro o método de la superclase, se puede referir a estos utilizando `base` o `MyBase` (C# o VB) con notación de punto. De esta manera se pueden acceder miembros con visibilidad en la superclase, ya sean atributos o métodos.

Otra cosa que llama la atención en el código es el uso de la palabra clave `new` o `Shadows` (C# o VB) en la declaración del atributo de OtraClase. Esto se debe a que se utiliza la palabra clave para indicar que se oculta la declaración de la superclase. Si se omite esta declaración, una advertencia indica que se debe utilizar, pero no afecta a la ejecución del código porque este es el comportamiento por defecto.

Muchas veces la rescritura es una herramienta eficaz para agregar funcionalidad en los servicios que presta una clase. De esta manera, con las técnicas explicadas, se escribe un método en una superclase que al rescribirlo en la subclase y ser más específico, agrega comportamiento a la misma, como se muestra a continuación.

Ejemplo

C#

```
namespace rescritura
{
    class Empleado
    {
        private String nombre;
        private double salario = 1500.00;
        private Fecha fechaNacimiento;

        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;
            }
        }
        public Fecha FechaNacimiento
        {
            get
            {
                return fechaNacimiento;
            }
            set
            {
                fechaNacimiento = value;
            }
        }
    }
}
```

VB

```
Public Class Empleado
    Private Dim _nombre As String
    Private Dim _salario As Double = 1500.0
    Private Dim _fechaNacimiento As Fecha

    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
    Public Property FechaNacimiento() As Fecha
        Get
            Return _fechaNacimiento
        End Get
        Set(ByVal value As Fecha)
            _fechaNacimiento = value
        End Set
    End Property
End Class
```

Si se desea que una clase derivada de Empleado brinde más detalles que los que da el método getDetalles, se incorpora dicho comportamiento como se muestra a continuación.

Ejemplo

C#

```
namespace rescritura
{
    class Gerente : Empleado
    {
        private String departamento = "legales";

        public override String getDetalles()
        {
            return base.getDetalles() + "\nDepartamento: " + departamento;
        }
        public String Departamento
        {
            get
            {
                return departamento;
            }
            set
            {
                departamento = value;
            }
        }
    }
}
```

VB

```
Public Class Gerente
    Inherits Empleado
    Private _departamento As String

    Public Overrides Function getDetalles() As String
        Return MyBase.getDetalles() + "\nDepartamento: " + Departamento
    End Function
```

```
    Public Property Departamento() As String
        Get
            Return _departamento
        End Get
        Set(ByVal value As String)
            _departamento = value
        End Set
    End Property
End Class
```

Métodos y propiedades que no se pueden sobrescribir en una subclase

Una subclase no puede sobrescribir métodos o propiedades que hayan sido declarados como `sealed override` o `NotOverridable` (C# o VB) en la superclase (por definición, estos métodos o propiedades no pueden ser sobrescritos). Si se intenta sobrescribir un método o propiedad `sealed override` o `NotOverridable` (C# o VB), el compilador mostrará un mensaje de error y no compilará el programa.

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 (forma en la cual se llama a los métodos `static` o `Shared` que se verán posteriormente).

Nota: Las estructuras son tipos `sealed` o `Shared` implícitamente, por consiguiente, no se pueden heredar.

La palabra clave `MyClass` *de VB*

La palabra clave `MyClass` se comporta como una variable de referencia que apunta a un objeto de la instancia actual de una clase como se había implementado originalmente. Es decir, si la clase pertenece a una cadena de herencia y es heredara por otras, cuando se declaren objetos en las subclases y se rescriban métodos o propiedades, en tiempo de ejecución dichos métodos y propiedades de la superclase quedarán ocultos porque Me tiene la referencia al objeto que se encuentra actualmente en ejecución. `MyClass` da acceso sobre este "ocultamiento" como si no existiese (`Me`, como tiene la referencia al objeto ejecutado actualmente debe resolver visibilidad para acceder). `MyClass` se parece a `Me`, pero cada método y propiedad llamada en `MyClass` es tratada como si el método o la propiedad fuesen declaradas como `NotOverridable`. Por lo tanto, el método o propiedad no se ve afectada por rescrituras en una clase derivada.

- `MyClass` es una palabra clave, no un objeto real. `MyClass` no se puede asignar a una variable, pasar a procedimientos, o utilizar en una comparación del tipo `Is`.
- `MyClass` se refiere a la clase en la que se utiliza y los miembros heredados por esta.
- `MyClass` puede utilizarse como calificador para los miembros `Shared`.

- `MyClass` no se puede utilizar dentro de un método `Shared`, pero se puede utilizar dentro de un método de instancia para acceder a un miembro `Shared` de una clase.
- `MyClass` no se puede usar en módulos estándar.
- `MyClass` puede utilizarse para calificar a un método que se define en una clase base y que no tiene ninguna implementación del método proporcionado en esa clase. Tal referencia tiene el mismo significado que `MyBase`.Método.

Ejemplo

VB

```
Public Class ClaseBase
    Public Overridable Sub MetodoDeVerificacion()
        Console.WriteLine("Mensaje en la superclase")
    End Sub
    Public Sub UsarMe()
        ' La siguiente llamada usa el método de la
        ' clase que creo el objeto actual. Si el método
        ' esta rescrito en la subclase, llamará a ese método
        Me.MetodoDeVerificacion()
    End Sub
    Public Sub UsarMyClass()
        ' El siguiente llamado se realiza al método
        ' implementado en esta clase, sin importar
        ' si fue rescrito o no. Supone al método
        ' de esta clase declarado como  NotOverridable
        MyClass.MetodoDeVerificacion()
    End Sub
End Class

Public Class ClaseDerivada : Inherits ClaseBase
    Public Overrides Sub MetodoDeVerificacion()
        Console.WriteLine("Mensaje en la subclase")
    End Sub
End Class

Public Class VerificarClases
    Sub Comienzo()
        Dim obj As ClaseDerivada = New ClaseDerivada()
        Console.WriteLine("Invocando al método con Me")
        obj.UsarMe()
        Console.WriteLine("Invocando al método con MyClass")
        obj.UsarMyClass()
        Console.ReadKey()
    End Sub
End Class

Module Module1

    Sub Main()
        Dim t As New VerificarClases()
        t.Comienzo()
        Console.ReadKey()
    End Sub
```

```
End Module
```

La salida producida por el programa es:

```
Invocando al método con Me
Mensaje en la subclase
Invocando al método con MyClass
Mensaje en la superclase
```

Métodos que una subclase debe sobrescribir

Las subclases deben sobrescribir aquellos métodos que hayan sido declarados como `abstract` o `MustOverride` (C# o VB) en la superclase, o la propia subclase debe ser abstracta. Escribir clases y métodos abstractos se explica con más detalle posteriormente.

Sobrescribir un método abstracto significa darle funcionalidad a un método que al declararse abstracto no tiene bloque de sentencias y se lo considera "incompleto". Notar que cuando una clase posee un método abstracto debe declarase ella también con la palabra clave `abstract` o `MustOverride` (C# o VB).

Ejemplo

C#

```
namespace abstractos
{
    public abstract class Empleado
    {
        protected String nombre="Alberto";
        protected double salario = 1500.00;
        protected Fecha fechaNacimiento;

        public abstract String getDetalles();
    }
}
```

VB

```
Public MustInherit Class Empleado
    Protected _nombre As String = "Alberto"
    Protected _salario As Double = 1500.0
    Protected _fechaNacimiento As Fecha

    Public MustOverride Function getDetalles() As String

End Class
```

Notar que la subclase incluye la palabra `override`.

Ejemplo

C#

```
namespace abstractos
{
    class Gerente : Empleado
    {
        private String departamento="Ventas";

        public override String getDetalles()
        {
            return "Nombre: " + nombre + "\nSalario: " + salario
                + "\nDep: " + departamento;
        }
    }
}
```

VB

```
Public Class Gerente
    Inherits Empleado

    Private _departamento As String = "Ventas"

    Public Overrides Function getDetalles() As String
        Return "Nombre: " + _nombre + _
            Environment.NewLine + "Salario: " + _salario.ToString + _
            Environment.NewLine + "Dep: " + _departamento
    End Function
End Class
```

Constructores en la herencia

Los constructores no se heredan, son propios de cada clase, lo cual implica que la construcción de un objeto es pura responsabilidad del tipo al que pertenece (clase).

Una subclase hereda métodos y atributos de la superclase que se podrán acceder por visibilidad directa o resolviéndola con `base` o `Mybase` (C# o VB). Esta última sentencia también se puede invocar en un constructor, donde este se utiliza con paréntesis, o mejor dicho, con el operador de llamado a función que invoca al constructor de la superclase. En C# esta declaración se realiza luego de poner el operador "**:**" luego del prototipo del constructor de la subclase y antes de la llave que abre el bloque de sentencias asociado al método. En VB tiene que ser la primera sentencia dentro del constructor y se utiliza en conjunción con el llamado a New (por ejemplo, `Mybase`.New).

Se deberá acceder al constructor de la clase base siempre desde la subclase, salvo que exista un constructor por defecto (sin argumentos) en cuyo caso se accede a él automáticamente. Si no se coloca explícitamente la invocación al constructor de la superclase, la invocación automática siempre es hacia un constructor por defecto. Si la superclase no posee un constructor por defecto, se obtendrá un error.

Cuando se construye un objeto de una subclase, se debe invocar al constructor de la superclase y pasarle los parámetros necesarios para la inicialización que esta requiera. Por ejemplo, el siguiente código invoca en la construcción de un objeto de tipo Gerente al constructor adecuado de tipo Empleado, pasándole el parámetro necesario para la construcción de la superclase.

Ejemplo

C#

```
namespace herenciaConstructores
{
    class Empleado
    {
        private String nombre;
        private double salario = 1500.00;
        private Fecha fechaNacimiento;
        public Empleado(String n, Fecha fDN)
        {
            // Llamado implícito a base;
            nombre = n;
            fechaNacimiento = fDN;
        }
        public Empleado(String n)
            : this(n, null)
        {

        }
    }
}
```

```
namespace herenciaConstructores
{
    class Gerente : Empleado
    {
        private String departamento;
        public Gerente(String n, String d): base(n)
        {
            departamento = d;
        }
    }
}
```

VB

```
 Public Class Empleado
    Private nombre As String
    Private fechaNacimiento As Fecha
    Private salario As Double

    Public Sub New(ByVal n As String, ByVal fDN As Fecha)
        nombre = n
        fechaNacimiento = fDN
```

```
    End Sub

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


Public Class Gerente
    Inherits Empleado

    Private departamento As String

    Public Sub New(ByVal n As String, ByVal d As String)
        MyBase.New(n)
        departamento = d
    End Sub
End Class
```

Existen situaciones en las cuales una superclase puede ser construida de diversas maneras. Es responsabilidad de la subclase utilizar el constructor apropiado en cada oportunidad. Si una superclase tiene una diversidad de constructores, la subclase deberá seleccionar entre ellos el que se ajuste al tipo de construcción que realice para los objetos de su tipo (el de la subclase). Por ejemplo, si se tiene la siguiente superclase:

Ejemplo

C#

```
namespace herenciaConstructoresThis
{
    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)
        {

        }
        // más código de Empleado...
    }
}
```

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
End Class
```


Una subclase para construir un objeto que tenga como superclase a la anterior, puede seleccionar los constructores de la superclase a través de los parámetros con que se invoque al usar `base` o `Mybase` (C# o VB). Si la cantidad de parámetros o el tipo de los mismos no se puede asociar a un constructor de la superclase, derivará en un error en tiempo de compilación, por ejemplo:

Ejemplo

C#

```
namespace herenciaConstructoresThis
{
    class Gerente : Empleado
    {
        private String departamento;
```

```
        public Gerente(String nombre, double salario, String dep)
            : base(nombre, salario)
        {
            departamento = dep;
        }
        public Gerente(String n, String d)
            : base(n)
        {
            departamento = d;
        }
        //    public Gerente(String dep) { // Error, no existe Empleado()
        //           departamento = dep;
        //    }
    }
}
```

VB

```
Public Class Gerente
    Inherits Empleado
    Private departamento As String

    Public Sub New(ByVal nombre As String, ByVal salario As Double, ByVal dep As
String)
        MyBase.New(nombre, salario)
        departamento = dep
    End Sub

    Public Sub New(ByVal n As String, ByVal d As String)
        MyBase.New(n)
        departamento = d
    End Sub

    'Public Sub New(ByVal n As String) ' Error, no existe New() en Empleado
    '    departamento = d
    'End Sub
End Class
```

En resumen, los objetos se construyen e inicializan según los llamados a los constructores en una cadena de herencia y la manera en que estos construyan cada respectiva superclase. El proceso completo de construcción e inicialización se puede resumir en lo siguiente:

- Se aloja memoria y ocurren las inicializaciones por defecto
- La inicialización de una variable de referencia utiliza estos pasos recursivamente cuando se construye una cadena de herencia:
  1. Enlazar los parámetros del constructor
  2. Si se llama explícitamente a `this`() o `Me.New()` (C# o VB) realizarlo recursivamente y luego saltar al paso 5
  3. Llamar recursivamente las invocaciones a `base`() o `MyBase.New()` (C# o VB) explícita o implícitas
  4. Ejecutar las inicializaciones explícitas de las variables de instancia
  5. Ejecutar el bloque de sentencias dentro del constructor actual

Niveles de acceso

Los niveles de acceso a una clase definen la visibilidad que tiene el código cliente (aquel que está fuera de la clase y no es un subtipo o tipo derivado) como así también de los subtipos de las clases (aquellos que son derivados de esta). Para especificar los niveles de accesibilidad se pueden utilizar los modificadores de acceso de guiándose con la siguiente tabla:

Modificador de Acceso	Significado
`public`	El acceso no está restringido. Se accede por notación de puntos desde cualquier código cliente
`protected`	El acceso está limitado a la clase donde se realiza la declaración o tipos derivados (subtipos) de dicha clase. No se accede por notación de puntos desde ningún código cliente.
`internal`	El acceso está limitado al ensamblado actual.
`protected internal`	El acceso está limitado al ensamblado actual o a los tipos derivados de la clase donde se realiza la declaración. Se accede por notación de puntos desde cualquier código cliente que este en el mismo ensamblado
`private`	El acceso está limitado a la clase donde se realiza la declaración. No se accede por notación de puntos desde ningún código cliente

Las accesibilidades declaradas mediante los modificadores establecen la visibilidad de los miembros declarados en una clase, con lo cual se debe tener en cuenta:

Sólo un modificador de acceso es permitido para un miembro o tipo, excepto cuando se utiliza la combinación `protected internal` o `Protected Friend` (C# o VB).

Los modificadores de acceso no están permitidos en los espacios de nombres. Los espacios de nombres no tienen restricciones de acceso.

Dependiendo del contexto en que se produce una declaración del elemento miembro, sólo ciertas accesibilidades declaradas están permitidas. Si no hay ningún modificador de acceso se especifica en una declaración de elemento miembro, se utiliza la accesibilidad por defecto.

Aquellos tipos de nivel superior, los que no están anidados en otros tipos (tipos anidados se verá en el capítulos de conceptos avanzados de clases), sólo pueden tener acceso `internal` o `public` ó `Friend` o `Public` (C# ó VB). La accesibilidad por defecto para estos tipos es `internal` o `Friend` (C# o VB).

La siguiente tabla es una guía de los niveles de acceso permitidos por cada modificador:

Modificador	Misma Clase	Mismo Ensamblado	Subclase	Código Cliente

private	Si	No	No	No
internal	Si	Si	No	No
protected	Si	No	Si	No
protected internal	Si	Si	Si	No
public	Si	Si	Si	Si

La segunda columna indica si la propia clase tiene acceso al miembro definido por el modificador de acceso. La tercera columna indica si las clases del mismo ensamblado que la clase. La cuarta columna indica si las subclases de la clase (sin importar dentro de que ensamblado o espacio de nombres se encuentren estas) tienen acceso a los miembros. La quinta columna indica si todas las clases tienen acceso a los miembros.

Los operadores de instancia

Estos operadores sólo pueden usarse con variables de referencia a un objeto. El objetivo de los mismos es determinar si un objeto es de un tipo determinado.

Por tipo se entiende a clase o interfaz (interface), es decir si responde a las palabras calificadoras "es un" para esa clase o interfaz, especificado a la derecha del operador o es del mismo tipo, según el operador utilizado.

Este tipo de operadores son muy específicos de los lenguajes y se explicarán por separado.

Los operadores `is` y `typeof` de C#

El operador `typeof` comprueba si un objeto es del mismo tipo que otro. Por ejemplo, si un objeto pertenece a una instancia de una clase dentro de la cadena de herencia de la clase que se coloca a la derecha del operador, el resultado booleano será verdadero.

Ejemplo

C#

```
        public void ConIs()
        {
            Empleado empleado = new Empleado("Alejandro", "Ignacio", "Fursi",
                    "17.767.076", "Ventas", "0", "GV50", 5500.0F);
            Gerente gerente = new Gerente("Juan", "Pedro", "Goyena", "17.767.076",
                    "Ventas", "0", "GV50", 5500.0F, 20);
            Director director = new Director("Daniel", "Federico", "Ruiz",
                    "17.767.076", "Ventas", "0", "GV50", 5500.0F, 20, "vocal");
            Secretaria secretaria = new Secretaria("María", "Juana", "Roldán",
                    "20.202.020", "Ventas", "20", "10", 800.0F, "inglés");
            Ingeniero ingeniero = new Ingeniero("Marío", "Alberto", "Rojas",
                    "20.333.025", "Producción", "20", "10", 2800.0F, "desarrollo");

            if (empleado is Empleado)
                Console.WriteLine("Empleado es del tipo Empleado");
            if (empleado is Persona)
```

```
            Console.WriteLine("Empleado es del tipo Persona");
        if (secretaria is Empleado)
            Console.WriteLine("Secretaria es del tipo Empleado");
        if (gerente is Empleado)
            Console.WriteLine("Gerente es del tipo Empleado");
        if (director is Empleado)
            Console.WriteLine("Director es del tipo Empleado");
        if (director is Empleado)
            Console.WriteLine("Director es del tipo Empleado");
        if (director is Persona)
            Console.WriteLine("Director es del tipo Persona");

        if (ingeniero is Gerente)
            Console.WriteLine("ingeniero es una instancia de Gerente");
        else
            Console.WriteLine("ingeniero NO es una instancia de Gerente");
    }
```

Las salidas que produce este método cuando se lo ejecuta son las siguientes:

```
Usando el operador is
Empleado es del tipo Empleado
Empleado es del tipo Persona
Secretaria es del tipo Empleado
Gerente es del tipo Empleado
Director es del tipo Empleado
Director es del tipo Empleado
Director es del tipo Persona
Ingeniero NO es una instancia de Gerente
```

Notar que cuando el objeto pertenece a un tipo dentro de la misma cadena de herencia se ejecuta la salida por verdadero en el condicional.

El operador `is` se utiliza para comprobar si el tipo en forma dinámica en tiempo de ejecución de un objeto es compatible con un tipo dado. El resultado de la operación e `is` T , donde e es una expresión y T es un tipo, es un valor booleano que indica si se puede convertir con éxito e al tipo T por una conversión de referencia, una conversión de boxing, o una conversión unboxing. La operación se evalúa como sigue:

1. Si el tipo en tiempo de compilación de e es el mismo que T, o si una conversión de referencia implícita o conversión boxing existe desde el tipo e a T en tiempo de compilación teniendo en cuenta:
    - Si e es un tipo por referencia, el resultado de la operación es equivalente a evaluar e != null .
    - Si e es de un tipo por valor, el resultado de la operación es `true` .
2. Si existe una conversión de referencia explícita o conversión unboxing desde el tipo en tiempo de compilación de e a T, se realiza una comprobación de tipo dinámico
    - Si el valor de e es `null` , el resultado es `false` .

- Caso contrario, considerar que R sea el tipo en tiempo de ejecución de la instancia de referencia de la expresión e. Si R y T son del mismo tipo y si R es un tipo por referencia y existe una conversión de referencia implícita a partir de R a T, o si R es un tipo de valor y T es un tipo interfaz que es implementada por R, el resultado es `true`.
- De lo contrario, el resultado es `false`.

3. Si ninguna situación anterior se ajusta a la evaluación y ninguna referencia o conversión boxing de e al tipo T es posible, y el resultado de la operación es `false`.

Nota: El `is` operador sólo tiene en cuenta las conversiones de referencias, las conversiones boxing y unboxing. Otras conversiones, como las conversiones definidas por el usuario, no son considerados por el operador `is`.

Para el uso correcto del operador tener en cuenta que:

- Una expresión con `is` da como resultado `true` si la expresión proporcionada no es nula, y el objeto siempre se puede convertir al tipo previsto sin provocar una excepción (condición de error).
- La palabra clave provoca una advertencia de tiempo de compilación si la expresión se sabe que es siempre verdadera o falsa, pero típicamente evalúa la compatibilidad de tipos en tiempo de ejecución.
- El operador `is` no se puede sobrecargar.

Cuando se quiere evaluar si un objeto es una instancia de una clase específica, se puede utilizar `typeof`. De manera diferente a `is`, este operador no evalúa si el tipo es convertible a uno dentro de la cadena de herencia o por medio de boxing o unboxing, simplemente retorna verdadero si un objeto es una instancia de una clase y falso en caso contrario.

Ejemplo

C#

```
        public void ConTypeof()
        {
            Empleado empleado = new Empleado("Alejandro", "Ignacio", "Fursi",
                    "17.767.076", "Ventas", "0", "GV50", 5500.0F);
            Gerente gerente = new Gerente("Juan", "Pedro", "Goyena", "17.767.076",
                    "Ventas", "0", "GV50", 5500.0F, 20);
            Director director = new Director("Daniel", "Federico", "Ruiz",
                    "17.767.076", "Ventas", "0", "GV50", 5500.0F, 20, "vocal");
            Secretaria secretaria = new Secretaria("María", "Juana", "Roldán",
                    "20.202.020", "Ventas", "20", "10", 800.0F, "inglés");
            Ingeniero ingeniero = new Ingeniero("Marío", "Alberto", "Rojas",
                    "20.333.025", "Producción", "20", "10", 2800.0F, "desarrollo");

            if (empleado.GetType() == typeof(Empleado))
                Console.WriteLine("Empleado es una instancia de Empleado");
            if (empleado.GetType() == typeof(Persona))
```

```
            Console.WriteLine("Empleado es una instancia de Persona");
        if (secretaria.GetType() == typeof(Empleado))
            Console.WriteLine("Secretaria es una instancia de Empleado");
        else if (secretaria.GetType() == typeof(Secretaria))
            Console.WriteLine("secretaria es una instancia de Secretaria");
        if (gerente.GetType() == typeof(Empleado))
            Console.WriteLine("Gerente es una instancia de Empleado");
        else if (gerente.GetType() == typeof(Gerente))
            Console.WriteLine("gerente es una instancia de Gerente");
        if (director.GetType() == typeof(Empleado))
            Console.WriteLine("Director es una instancia de Empleado");
        else if (director.GetType() == typeof(Director))
            Console.WriteLine("director es una instancia de Director");
        if (director.GetType() == typeof(Empleado))
            Console.WriteLine("Director es una instancia de Empleado");
        if (director.GetType() == typeof(Persona))
            Console.WriteLine("Director es una instancia de Persona");

        if (ingeniero.GetType() == typeof(Gerente))
            Console.WriteLine("ingeniero es una instancia de Gerente");
        else
            Console.WriteLine("ingeniero NO es una instancia de Gerente");
    }
```

El resultado de ejecutar este método en un programa son las siguientes salidas:

```
Usando el operador typeof
Empleado es una instancia de Empleado
secretaria es una instancia de Secretaria
gerente es una instancia de Gerente
director es una instancia de Director
ingeniero NO es una instancia de Gerente
```

El operador as de C#

El operador as trabaja como una operación de conversión de tipo con la salvedad que retorna null y en lugar de lanzar una excepción cuando no puede realizar la conversión

Ejemplo

C#

```
    public void ConAs()
    {
        Persona p;
        Empleado empleado = new Empleado("Alejandro", "Ignacio", "Fursi",
                "17.767.076", "Ventas", "0", "GV50", 5500.0F);
        Gerente gerente = new Gerente("Juan", "Pedro", "Goyena", "17.767.076",
                "Ventas", "0", "GV50", 5500.0F, 20);
        Director director = new Director("Daniel", "Federico", "Ruiz",
                "17.767.076", "Ventas", "0", "GV50", 5500.0F, 20, "vocal");
        Secretaria secretaria = new Secretaria("María", "Juana", "Roldán",
                "20.202.020", "Ventas", "20", "10", 800.0F, "inglés");
        Ingeniero ingeniero = new Ingeniero("Marío", "Alberto", "Rojas",
                "20.333.025", "Producción", "20", "10", 2800.0F, "desarrollo");
```

```
            p = empleado as Persona;
            Console.WriteLine(p.Apellido + ", " + p.PrimerNombre+ " " +
                              p.SegundoNombre);

            p = gerente as Persona;
            Console.WriteLine(p.Apellido + ", " + p.PrimerNombre + " " +
                              p.SegundoNombre);

            p = director as Persona;
            Console.WriteLine(p.Apellido + ", " + p.PrimerNombre + " " +
                              p.SegundoNombre);

            p = secretaria as Persona;
            Console.WriteLine(p.Apellido + ", " + p.PrimerNombre + " " +
                              p.SegundoNombre);

            p = ingeniero as Persona;
            Console.WriteLine(p.Apellido + ", " + p.PrimerNombre + " " +
                              p.SegundoNombre);
        }
```

El resultado de ejecutar este método en un programa son las siguientes salidas:

```
Usando el operador as
Fursi, Alejandro Ignacio
Goyena, Juan Pedro
Ruiz, Daniel Federico
Roldán, María Juana
Rojas, Marío Alberto
```

El operador `TypeOf ... Is` de VB

Comprueba si un objeto es del mismo tipo que otro (compatible). Por ejemplo, si un objeto pertenece a una instancia de una clase dentro de la cadena de herencia de la clase que se coloca a la derecha del operador, el resultado booleano será verdadero. Suponiendo la siguiente expresión, se analiza su evaluación:

```
TypeOf ExpresiónObjeto Is Tipo
```

Donde:

- ExpresiónObjeto: es cualquier expresión que retorne una referencia válida a un objeto (no nula). Si el resultado de la expresión es `Nothing`, retorna `False` siempre.
- Tipo: es un tipo válida declarado por el usuario o un tipo del Framework de .Net

El operador `TypeOf` determina si el tipo en tiempo de ejecución de ExpresiónObjeto es compatible con Tipo. La compatibilidad depende de las cadenas de herencia declaradas. La tabla siguiente muestra cómo se determina la compatibilidad.

Categoría Tipo	Criterio de Compatibilidad
Clase	*ExpresiónObjeto* es del *Tipo* o hereda de *Tipo*

Estructura	*ExpresiónObjeto* es del *Tipo* o implementa una interfaz
Interfaz	*ExpresiónObjeto* implementa la interfaz como *Tipo* o hereda de una clase que implementa la interfaz

Ejemplo

VB

```
    Public Sub ConTypeOfIs()
        Dim _empleado As New Empleado("Alejandro", "Ignacio", "Fursi", _
                              "17.767.076", "Ventas", "0", "GV50", 5500.0F)
        Dim _gerente As New Gerente("Juan", "Pedro", "Goyena", _
                             "17.767.076", "Ventas", "0", "GV50", 5500.0F, 20)
        Dim _director As New Director("Daniel", "Federico", "Ruiz", _
                           "17.767.076", "Ventas", "0", "GV50", 5500.0F, 20, "vocal")
        Dim _secretaria As New Secretaria("María", "Juana", "Roldán", _
                        "20.202.020", "Ventas", "20", "10", 800.0F, "inglés")
        Dim _ingeniero As New Ingeniero("Marío", "Alberto", "Rojas", _
                        "20.333.025", "Producción", "20", "10", 2800.0F, "desarrollo")

        If (TypeOf Nothing Is Empleado) Then
            Console.WriteLine("Empleado es del tipo Empleado")
        End If

        If (TypeOf _empleado Is Persona) Then
            Console.WriteLine("Empleado es del tipo Persona")
        End If

        If (TypeOf _secretaria Is Empleado) Then
            Console.WriteLine("Secretaria es del tipo Empleado")
        End If

        If (TypeOf _gerente Is Empleado) Then
            Console.WriteLine("Gerente es del tipo Empleado")
        End If

        If (TypeOf _director Is Empleado) Then
            Console.WriteLine("Director es del tipo Empleado")
        End If

        If (TypeOf _director Is Empleado) Then
            Console.WriteLine("Director es del tipo Empleado")
        End If

        If (TypeOf _director Is Persona) Then
            Console.WriteLine("Director es del tipo Persona")
        End If

        ' Esto es un error porque nunca un ingeniero puede ser un Gerente
        'If (TypeOf _ingeniero Is Gerente) Then
        '    Console.WriteLine("ingeniero es una instancia de Gerente")
        'Else
```

```
        '    Console.WriteLine("ingeniero NO es una instancia de Gerente")
        'End If
    End Sub
```

El resultado de ejecutar este método en un programa son las siguientes salidas:

```
Usando el operador TypeOf XXX Is
Empleado es del tipo Persona
Secretaria es del tipo Empleado
Gerente es del tipo Empleado
Director es del tipo Empleado
Director es del tipo Empleado
Director es del tipo Persona
```

El operador `Is` de VB

Se utiliza para comparar dos variables de referencia a objetos. El operador `Is` determina si dos referencias a objetos se apuntan al *mismo objeto*. Se debe tener cuidado porque no lleva a cabo ninguna comparación entre los valores de los atributos que posean dichos objetos. Si tanto objeto1 como objeto2 se refieren a la misma instancia de objeto exactamente, el resultado es `True`, y si no lo hacen, el resultado es `False`.

También se puede utilizar con la palabra clave `TypeOf` para hacer una expresión `TypeOf ... Is`, que comprueba si una variable de objeto es compatible con un tipo de datos (esto es, que se encuentra en la misma cadena de herencia).

Ejemplo

VB

```
    Public Sub ConIs()
        Dim _empleado1 As New Empleado( _
              "Alejandro", "Ignacio", "Fursi", "17.767.076", _
              "Ventas", "0", "GV50", 5500.0F)
        Dim _gerente1 As New Gerente( _
              "Juan", "Pedro", "Goyena", "17.767.076", _
              "Ventas", "0", "GV50", 5500.0F, 20)
        Dim _director1 As New Director( _
              "Daniel", "Federico", "Ruiz", "17.767.076", _
              "Ventas", "0", "GV50", 5500.0F, 20, "vocal")
        Dim _secretaria1 As New Secretaria( _
              "María", "Juana", "Roldán", "20.202.020", _
              "Ventas", "20", "10", 800.0F, "inglés")
        Dim _ingeniero1 As New Ingeniero( _
              "Marío", "Alberto", "Rojas", "20.333.025", _
              "Producción", "20", "10", 2800.0F, "desarrollo")

        Dim _empleado2 As Empleado = _empleado1
        Dim _gerente2 As Gerente = _gerente1
        Dim _director2 As Director = _director1
        Dim _secretaria2 As Secretaria = _secretaria1
        Dim _ingeniero2 As Ingeniero = _ingeniero1
```

```
        If _empleado1 Is _empleado2 Then
            Console.WriteLine( _
              "_empleado1 apunta a la misma instancia que _empleado2")
        End If
        If _empleado1 Is _gerente1 Then
            Console.WriteLine( _
              "_empleado1 apunta a la misma instancia que _gerente1")
        Else
            Console.WriteLine( _
              "_empleado1 NO apunta a la misma instancia que _gerente1")
        End If

        If _secretaria1 Is _empleado1 Then
            Console.WriteLine( _
              "_secretaria1 apunta a la misma instancia que _empleado1")
        ElseIf _secretaria1 Is _secretaria2 Then
            Console.WriteLine( _
              "_secretaria1 apunta a la misma instancia que _secretaria2")
        End If
        If _gerente1 Is _empleado1 Then
            Console.WriteLine( _
              "_gerente1 apunta a la misma instancia que _empleado1")
        ElseIf _gerente1 Is _gerente2 Then
            Console.WriteLine( _
              "_gerente1 apunta a la misma instancia que _gerente2")
        End If
        If _director1 Is _empleado1 Then
            Console.WriteLine( _
              "_director1 apunta a la misma instancia que _empleado1")
        ElseIf _director1 Is _director2 Then
            Console.WriteLine( _
              "_director1 apunta a la misma instancia que _director2")
        End If
        If (_director1 Is (_empleado1)) Then
            Console.WriteLine("Director es una instancia de Empleado")
        End If

        If _ingeniero1 Is _gerente1 Then
            Console.WriteLine( _
              "_ingeniero1 apunta a la misma instancia que _gerente1")
        Else
            Console.WriteLine( _
              "_ingeniero1 NO apunta a la misma instancia que _gerente1")
        End If
    End Sub
```


El resultado de ejecutar este método en un programa son las siguientes salidas:

```
Usando el operador XXX Is YYY
_empleado1 apunta a la misma instancia que _empleado2
_empleado1 NO apunta a la misma instancia que _gerente1
_secretaria1 apunta a la misma instancia que _secretaria2
_gerente1 apunta a la misma instancia que _gerente2
_director1 apunta a la misma instancia que _director2
```

```
_ingeniero1 NO apunta a la misma instancia que _gerente1
```