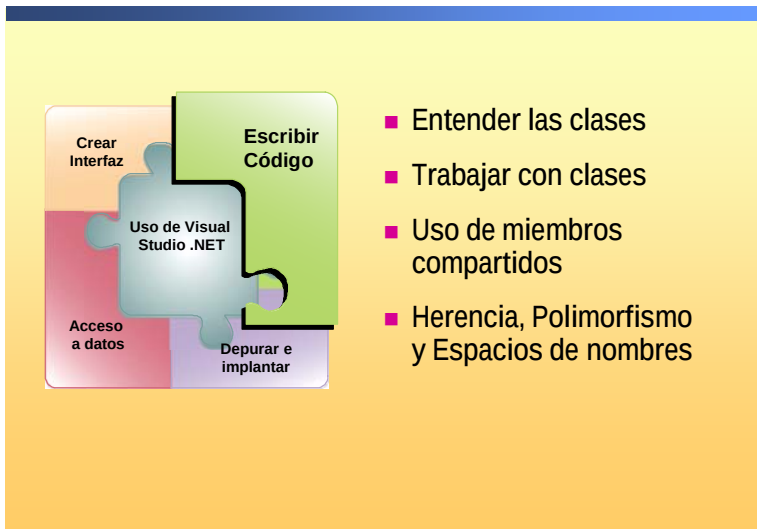


Programación orientada a objetos en Visual Basic .NET

Índice

Introducción	1
Lección: Entender el concepto de clase	2
Lección: Trabajar con clases	8
Lección: Uso de miembros compartidos	22
Lección: herencia, polimorfismo y espacios de nombres	28

Introducción



Introducción

Este módulo explica cómo crear y utilizar clases. El módulo describe los conceptos de abstracción, encapsulación, instanciación, inicialización, constructores y destructores. Este módulo también define la herencia, el polimorfismo y los espacios de nombres.

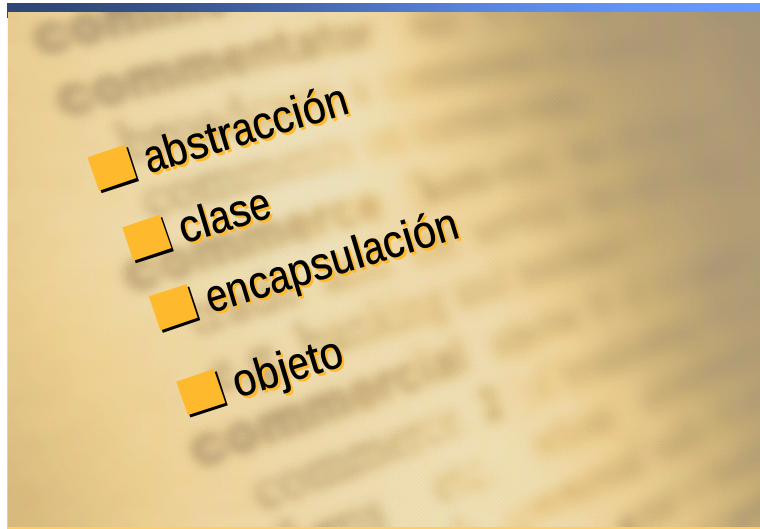
Los objetos son fundamentales para la programación en Visual Basic® .NET. Los formularios, controles y bases de datos son objetos. En este módulo, aprenderemos cómo crear nuestros propios objetos desde clases definidas por nosotros y cómo podemos utilizar objetos para simplificar la programación e incrementar la reutilización de código.

Objetivos

En este módulo, aprenderemos a:

- Entender los conceptos de programación orientada a objetos, incluyendo clase, abstracción, encapsulación y objeto.
- Utilizar el Examinador de objetos para examinar los elementos de programación disponibles, incluyendo clases y objetos.
- Crear una nueva clase, incluyendo sus métodos, propiedades y miembros de datos.
- Crear y utilizar la instancia de una clase, incluyendo miembros de instancia y de datos compartidos, y métodos compartidos y no compartidos.
- Explicar cómo funcionan los constructores y los destructores.
- Explicar la herencia, el polimorfismo y los espacios de nombres.

Lección: Entender el concepto de clase



Introducción

Esta lección describe los conceptos de programación orientada a objetos básicos, incluyendo clases, abstracción y encapsulación, y objetos. Entender estos conceptos nos permite utilizar las potentes características orientadas a objetos de Visual Basic .NET. Además, esta lección describe cómo utilizar el Examinador de objetos para examinar clases, objetos y otros elementos de programación.

Estructura de la lección

Esta lección incluye los siguientes temas:

- ¿Qué es una clase?
- ¿Qué es un objeto?
- Cómo utilizar el Examinador de objetos

Objetivos de la lección

En esta lección, aprenderemos a:

- Describir clases y objetos.
- Explicar el papel de la abstracción y encapsulación en la creación de clases.
- Utilizar el Examinador de objetos para examinar clases, objetos y demás elementos de programación.

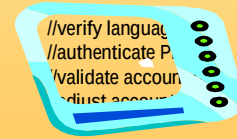
¿Qué es una clase?

- Una *clase* es un blueprint que describe un objeto y define atributos y operaciones para el objeto
- Las clases utilizan la *abstracción* para poner a disposición únicamente los elementos esenciales para definir el objeto
- Las clases utilizan la *encapsulación* para hacer cumplir una abstracción

Lo que ve el usuario:



Lo que está encapsulado:



Introducción

En la programación orientada a objetos, utilizamos la abstracción y la encapsulación para crear clases bien diseñadas.

Definición

Una *clase* es una plantilla o una estructura preliminar de un objeto. Esta estructura preliminar define atributos para almacenar datos y define operaciones para manipular esos datos. Una clase también define un conjunto de restricciones para permitir o denegar el acceso a sus atributos y operaciones.

Uso de la abstracción

Para crear una clase bien diseñada, utilizaremos la *abstracción*. Al implementar la abstracción, definiremos un concepto utilizando un mínimo conjunto de funcionalidades pensadas cuidadosamente que proporcione el comportamiento fundamental de la clase de un modo fácil de utilizar. Por desgracia, no es fácil crear buenas abstracciones de software. Normalmente, encontrar buenas abstracciones requiere un profundo conocimiento del problema que ha de resolver la clase y su contexto, una gran claridad de ideas y mucha experiencia.

Ejemplo de abstracción

El formulario Visual Basic .NET con el que hemos estado trabajando es un buen ejemplo de abstracción. Las propiedades esenciales de un formulario, como el título y color de fondo, se han abstraído en la clase **Form**. Algunas operaciones esenciales que se han abstraído son abrir, cerrar y minimizar.

Uso de la encapsulación	La abstracción se garantiza mediante la <i>encapsulación</i> . La encapsulación es el empaquetamiento de atributos y funcionalidades para crear un objeto que esencialmente es una “caja negra” (cuya estructura interna permanece privada). Empaquetamos los detalles de la abstracción y proporcionamos acceso sólo a los elementos que necesitan estar accesibles. Otros objetos pueden acceder a los servicios de un objeto encapsulado únicamente mediante mensajes que pasan a través de una interfaz claramente definida.
Ejemplo de encapsulación	Un ejemplo de encapsulación es un terminal de autoservicio (<i>automatic teller machine</i> , ATM). La interfaz de la ATM es simple para el cliente ATM, y el funcionamiento interno está oculto. Del mismo modo, una clase BankAccount encapsularía los métodos, campos y propiedades que describen una cuenta bancaria. Sin la encapsulación, deberíamos declarar procedimientos y variables distintos para almacenar y gestionar información de la cuenta bancaria, y sería difícil trabajar con más de una cuenta bancaria a la vez. La encapsulación permite a los usuarios utilizar los datos y procedimientos de la clase BankAccount como una unidad, sin conocer el código concreto encapsulado en la clase.

¿Qué es un objeto?

- Un objeto es una instancia de una clase
- Los objetos tienen las siguientes cualidades:
 - Identidad: los objetos se distinguen uno de otro
 - Comportamiento: los objetos pueden realizar tareas
 - Estado: los objetos almacenan información que puede cambiar con el tiempo



Introducción

En general, para ejecutar los métodos y utilizar las propiedades de una clase, necesitamos crear una *instancia* de la clase. La instancia de una clase se denomina objeto. Las palabras *clase* y *objeto* se utilizan con tanta frecuencia en la programación orientada a objetos que es fácil confundir los términos. Piense en una clase como una representación abstracta de algo y en un objeto como en un ejemplo utilizable de lo que representa la clase.

Definición

Un *objeto* es una unidad de software que contiene una colección de métodos y datos relacionados. Un objeto es una instancia específica de una clase, e incluye las características de esa clase.

Ejemplo de objeto

La palabra *coche* tiene diferentes significados en contextos distintos. En ocasiones, hace referencia al concepto general de coche o al conjunto de todos los coches. El uso de la palabra de este modo es parecido al uso de la palabra *clase*. En ocasiones, la palabra *coche* se refiere a un coche específico. El uso de la palabra de este modo es parecido al uso de la palabra *objeto*.

Identidad

Los objetos tienen identidad. La *identidad* es la característica que distingue un objeto del resto.

Comportamiento

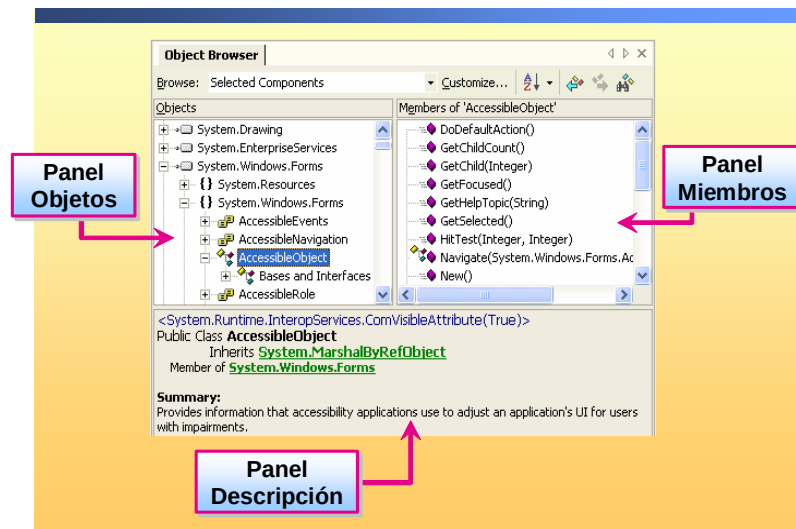
Los objetos proporcionan comportamiento. El *comportamiento* es la característica que hace útiles los objetos. Los objetos existen para proporcionar comportamiento. Por ejemplo, la mayor parte del tiempo podemos ignorar el funcionamiento interno de un coche y pensar en su comportamiento. Los coches son útiles porque podemos conducirlos. El funcionamiento interno existe, pero normalmente es inaccesible. Lo que es accesible es el comportamiento de un objeto. Los objetos de la misma clase comparten el mismo comportamiento. Un coche es un coche porque podemos conducirlo.

Estado

Los objetos tienen estado. El *estado* es la característica que hace referencia al funcionamiento interno de un objeto y que le permite proporcionar el comportamiento que le define. Un objeto bien diseñado mantiene su estado inaccesible. El estado está estrechamente relacionado con la abstracción y la encapsulación. No nos preocupa *cómo* realiza un objeto sus acciones, pero sí

que las realice. Dos objetos podrían contener casualmente el mismo estado, pero seguirían siendo dos objetos distintos.

Cómo utilizar el Examinador de objetos



Introducción

Podemos utilizar el Examinador de objetos para examinar los elementos de programación de un componente (espacios de nombres, clases, módulos, estructuras, etc.) y los miembros de esos elementos (propiedades, métodos, eventos, variables, etc.). Los componentes que examinemos pueden ser proyectos de nuestra solución, componentes referenciados en esos proyectos o componentes externos.

Uso del Examinador de objetos

Para abrir el Examinador de objetos, pulse F2 o, en el menú **Ver**, seleccione **Examinador de objetos**.

El Examinador de objetos está formado por tres paneles:

- El *panel Objetos* muestra todos los objetos contenedores del ámbito de búsqueda en una vista en árbol. Cuando expandimos un elemento haciendo doble clic en él o haciendo clic en el signo más (+) junto a su nombre, aparecen los elementos definidos en él.
- El *panel Miembros* muestra los miembros de un elemento cuando seleccionamos el elemento en el panel Objetos.
- El *panel Descripción* muestra información detallada sobre el elemento o miembro actualmente seleccionado.

Un icono específico representa cada elemento de programación en el Examinador de objetos. La siguiente tabla muestra algunos de los iconos que se utilizan:

Icono	Descripción	Icono	Descripción
{ }	Espacio de nombres		Módulo
	Clase		Estructura

Lección: Trabajar con clases

- Cómo crear una nueva clase
- Cómo agregar miembros de datos de una instancia
- Cómo agregar métodos
- Cómo agregar propiedades
- Cómo crear una instancia de una clase
- Cómo utilizar constructores
- Cómo utilizar destructores

Introducción

La plataforma Microsoft .NET proporciona la biblioteca de clases del .NET Framework, pero también nosotros podemos crear nuestras propias clases. Esta lección describe cómo crear una nueva clase, agregar miembros de datos, métodos y propiedades a la clase, y establecer modificadores de acceso públicos y privados. Esta lección también describe cómo crear una instancia de una clase y cómo inicializar y desinicializar objetos.

Estructura de la lección

Esta lección incluye los siguientes temas y actividades:

- Cómo crear una nueva clase
- Cómo agregar miembros de datos de una instancia
- Cómo agregar métodos
- Cómo agregar propiedades
- Cómo crear una instancia de una clase
- Práctica: creación de una clase
- Cómo utilizar los constructores
- Cómo utilizar los destructores

Objetivos de la lección

En esta lección, aprenderemos a:

- Crear una nueva clase.
- Agregar miembros de datos de una instancia a una nueva clase.
- Agregar métodos a una nueva clase.
- Agregar propiedades a una nueva clase.
- Crear un objeto.
- Utilizar constructores para inicializar nuevos objetos.
- Utilizar destructores para desinicializar objetos.

Cómo crear una nueva clase

- Crear una nueva clase utilizando el comando Agregar clase del menú Proyecto
- Ejemplo de una nueva clase denominada BankAccount:

```
Public Class BankAccount
End Class
```

Introducción

Una vez finalizado el proceso de abstracción para determinar las entidades relevantes de un problema de negocio determinado, estaremos preparados para crear clases que reflejen las entidades en la abstracción.

Crear una nueva clase

⚡ Crear una nueva clase

1. Abrir un proyecto en Visual Studio .NET (si no hay ninguno abierto).
2. En el menú **Proyecto**, hacer clic en **Agregar clase**.
3. En el cuadro **Agregar nuevo elemento**, escribir el nombre de la nueva clase y hacer clic en **Abrir**.

El Editor de código proporciona instrucciones de programación que marcan las instrucciones de inicio y final de la clase, como sigue:

```
Public Class ClassName
End Class
```

Cómo agregar miembros de datos de una instancia

- Agregar un miembro de datos denominado *balance*

```
Public Class BankAccount
    Private balance As Double
End Class
```

Palabra clave	Definición
Public	Accesible en todas partes
Private	Accesible sólo en el propio tipo
Protected	Accesible sólo por clases que heredan de la clase

Introducción

Tras agregar una nueva clase a nuestro proyecto, podemos agregar miembros de datos a la clase. Un miembro de datos específico para una instancia de la clase se denomina *miembro de datos de una instancia*. Cuando agregamos miembros de datos de una instancia a una clase, especificamos el nivel de acceso estableciendo los modificadores de acceso.

Miembros de datos de una instancia

Los miembros de datos de una instancia incluyen variables y constantes miembro. Las variables miembro también se denominan *campos*.

Ejemplo

En el siguiente ejemplo, se agrega a la clase un miembro de datos de tipo **Double** denominado *balance* (saldo):

```
Private balance As Double
```

Palabra clave Me

También podemos hacer referencia al miembro de datos *balance* utilizando la palabra clave **Me**, como se muestra en el siguiente ejemplo:

```
Me.balance
```

La palabra clave **Me** se comporta como una variable de objeto haciendo referencia a la instancia actual de una clase. Cuando una clase puede tener más de una instancia, la palabra clave **Me** proporciona una forma para hacer referencia a la instancia específica de la clase en la que el código se está ejecutando actualmente.

Modificadores de acceso

Podemos controlar la accesibilidad de las entidades dentro de esta clase: algunas estarán accesibles únicamente desde dentro y otras desde dentro y desde fuera. Los miembros de la entidad accesibles únicamente desde dentro son privados. Los miembros de la entidad accesibles tanto desde dentro como desde fuera son públicos. Por ejemplo, necesitaríamos hacer que el miembro de datos de la instancia *balance* (saldo) fuera privado para que el saldo de la cuenta sólo pudiera modificarse desde dentro de la clase **BankAccount**.

También podemos utilizar la palabra clave **Protected** para limitar la accesibilidad a clases que *hereden* de esta clase. Estudiaremos con más detalle la herencia en la última lección de este módulo.

Cómo agregar métodos

- Agregar un método denominado **Deposit**

```
Public Class BankAccount
    Private balance As Double

    Public Sub Deposit(ByVal amount As Double)
        balance += amount
    End Sub
End Class
```

- Métodos sobrecargados: dos o más métodos con el mismo nombre pero diferentes firmas
Ejemplo: `MessageBox.Show`

Introducción

Podemos agregar métodos a una clase. Cuando agregamos métodos, especificamos el nivel de acceso estableciendo el modificador de acceso. Los métodos incluyen funciones y procedimientos **Sub**.

Ejemplo

En el siguiente ejemplo, se agrega a la clase un método denominado **Deposit**:

```
Public Sub Deposit(ByVal amount As Double)
    Balance += amount
End Sub
```

En este ejemplo, el método **Deposit** es público, por tanto los usuarios de esta clase pueden depositar dinero en una cuenta utilizando el método **Deposit**.

Sobrecarga de métodos

La sobrecarga de métodos se produce cuando una clase contiene dos o más métodos con el mismo nombre pero distintas firmas. Un ejemplo de método sobrecargado es el método **MessageBox.Show**. El método **Show** proporciona doce sobrecargas. Estas sobrecargas hacen que el método sea más flexible para los usuarios del método.

Para sobrecargar un método, utilice la palabra clave **Overloads**. La palabra clave **Overloads** declara una propiedad o método con el mismo nombre que un miembro existente, pero con una lista de parámetros distinta del miembro original.

El siguiente ejemplo muestra un método sobrecargado denominado **Execute**. El primer método **Execute** no toma ningún parámetro. El segundo, únicamente toma como parámetro una cadena.

```
Public Overloads Sub Execute( )
    ...
End Sub
Public Overloads Sub Execute(ByVal connection As String)
    ...
End Sub
```

Cómo agregar propiedades

■ Agregar una propiedad:

```
Public Class BankAccount
    Private customerName As String

    Public Property Name( ) As String
        Get
            Return customerName
        End Get
        Set(ByVal Value As String)
            customerName = Value
        End Set
    End Property
End Class
```

Introducción

Podemos agregar propiedades a una nueva clase en nuestro proyecto. Podemos asignar la propiedad, y podemos recuperar el valor de la propiedad desde la clase.

Agregar propiedades

Para agregar miembros de propiedades a una clase, normalmente definimos un miembro de datos privado y procedimientos de propiedades públicos. Podemos realizar dos tipos de procedimientos en propiedades en Visual Basic .NET: **Get** y **Set**.

- El procedimiento **Get** recupera el valor de la propiedad desde la clase. No debería modificar el valor.
- El procedimiento **Set** asigna la propiedad.

Ejemplo

En el siguiente ejemplo, se agrega a la clase **BankAccount** una propiedad denominada **Name**:

```
Public Class BankAccount
    Private customerName As String

    Public Property Name( ) As String
        Get
            Return customerName
        End Get
        Set(ByVal Value As String)
            customerName = Value
        End Set
    End Property
End Class
```

ReadOnly y WriteOnly

Utilizamos el modificador **ReadOnly** en la declaración de la propiedad para crear únicamente la propiedad **Get**. Utilizamos el modificador **WriteOnly** en la declaración de la propiedad para crear únicamente la propiedad **Set**.

Cómo crear una instancia de una clase

- Uso de la palabra clave **New** para crear una instancia de la clase **BankAccount** :

```
Module Bank  
    Sub Main  
        Dim account As New BankAccount( )  
        account.Deposit(500.00)  
    End Sub  
End Module
```

Introducción Para ejecutar los métodos y utilizar las propiedades de una clase, debemos crear una instancia de la clase. La instancia de una clase se denomina objeto.

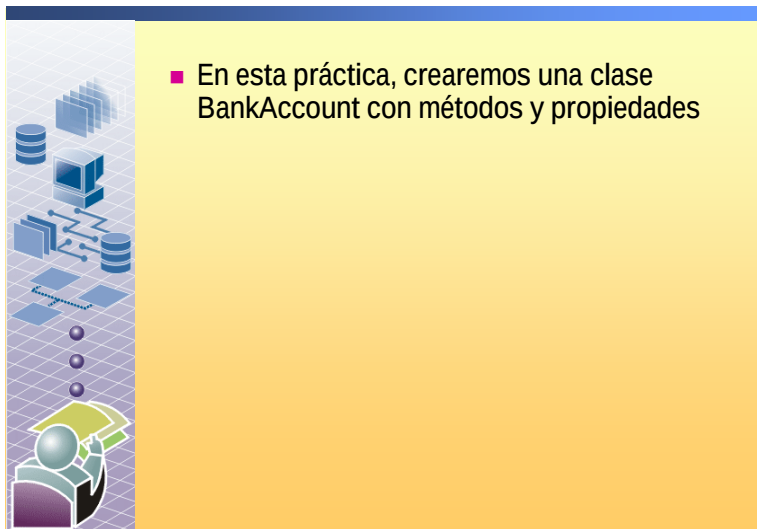
Sintaxis Para crear una instancia de una clase, declaramos una variable del tipo de la clase y utilizamos la palabra clave **New**, como se muestra en la siguiente línea de código:

```
Dim nombre_objeto As New tipo_objeto( )
```

Ejemplo El siguiente ejemplo muestra cómo crear una instancia de la clase **BankAccount** utilizando la palabra clave **New**:

```
Sub Main  
    Dim account As New BankAccount( )  
    account.Deposit(500.00)  
End Sub
```

Práctica: Creación de una clase



En esta práctica, crearemos una clase **BankAccount** con métodos y propiedades.

⚡ Agregar una nueva clase a un proyecto

1. Abrir una nueva aplicación Windows en Visual Basic .NET. Establecer **SimpleClass** como nombre del proyecto y ubicarlo en una carpeta llamada **CreatingClass**.
2. En el menú **Proyecto**, hacer clic en **Agregar clase**.
3. En el cuadro de diálogo **Agregar nuevo elemento**, cambiar el nombre de clase por **BankAccount.vb**, y hacer clic en **Abrir**.

⚡ Agregar miembros de datos, métodos y propiedades a la clase

1. Agregar un miembro de datos privado denominado *customerBalance* de tipo **Double**, como se muestra en la siguiente línea de código:

```
Private customerBalance As Double
```

2. Agregar otro miembro de datos privado denominado *customerName* de tipo **String**, como se muestra en la siguiente línea de código:

```
Private customerName As String
```

3. Crear un método público denominado **Deposit** que toma un parámetro **amount** de tipo **Double** por valor, como sigue:

```
Public Sub Deposit(ByVal amount As Double)
End Sub
```

4. En el método, incrementar el valor del saldo (balance) añadiéndole *amount*, como sigue:

```
customerBalance += amount
```

5. Crear una propiedad pública denominada **Name**, como sigue:

```
Public Property Name( ) As String  
  
End Property
```

6. En el bloque **Get** de la propiedad, devolver **customerName**, como sigue:

```
Get  
    Return customerName  
End Get
```

7. En el bloque **Set** de la propiedad, asignar **Value** a **customerName**, como sigue:

```
Set(ByVal Value As String)  
    customerName = Value  
End Set
```

8. Crear una propiedad de sólo lectura denominada **Balance** que devuelva el saldo actual, como sigue:

```
Public ReadOnly Property Balance( ) As Double  
    Get  
        Return customerBalance  
    End Get  
End Property
```

El código complete de la clase **BankAccount** debería ser como el siguiente:

```
Public Class BankAccount
    Private customerBalance As Double
    Private customerName As String

    Public Sub Deposit(ByVal amount As Double)
        customerBalance += amount
    End Sub

    Public ReadOnly Property Balance( ) As Double
        Get
            Return customerBalance
        End Get
    End Property

    Public Property Name( ) As String
        Get
            Return customerName
        End Get
        Set(ByVal Value As String)
            customerName = Value
        End Set
    End Property
End Class
```

⚡ Crear una utilidad de prueba para la clase **BankAccount**

1. Agregar al proyecto un modulo denominado TestHarness.vb.
2. Crear un procedimiento **Sub** público denominado **Main**, como sigue:

```
Public Sub Main( )
End Sub
```

3. Dentro del **Sub Main**, crear una instancia de la clase **BankAccount**, como sigue:

```
Dim account As New BankAccount( )
```

4. Utilizar la propiedad **Name** para asignar el nombre a la cuenta, como sigue:

```
account.Name = "Joe"
```

5. Utilizar el método **Deposit** para depositar dinero en la cuenta, como sigue:

```
account.Deposit(500)
```

6. Utilizar un cuadro de mensaje para mostrar el nombre de la cuenta y su saldo, como sigue:

```
MessageBox.Show ("Name: " & account.Name & _
    ". Balance: $" & account.Balance( ))
```

7. Cambiar el objeto de inicio del proyecto para utilizar **Sub Main**.
8. Generar y ejecutar el programa.

El código complete del módulo TestHarness debería ser como el siguiente:

```
Module TestHarness

Public Sub Main( )
    Dim account As New BankAccount( )

    account.Name = "Joe"
    account.Deposit(500)

    MessageBox.Show("Name: " & account.Name & _
        ". Balance: $" & account.Balance( ))

End Sub
End Module
```

Archivos de solución

Los archivos de solución de esta práctica se encuentran en la carpeta CreatingClass\Solution dentro del fichero practs10.zip.

Cómo utilizar los constructores

- Ejecutan código cuando el objeto está instanciado

```
Public Sub New( )
    ' Perform simple initialization
    value = 1
End Sub
```

- Pueden sobrecargarse, pero no utilizan la palabra clave Overloads

```
Public Sub New(ByVal i As Integer)
    ' Overloaded without Overloads keyword
    ' Perform more complex initialization
    value = i
End Sub
```

Introducción

En Visual Basic .NET, la inicialización de nuevos objetos se controla utilizando constructores. Para crear un constructor para una clase, se crea un procedimiento denominado **Sub New** en cualquier lugar de la definición de la clase.

Características de Sub New

El constructor **Sub New** tiene las siguientes características:

- El código del bloque **Sub New** siempre se ejecutará antes de cualquier otro código de una clase.
- El constructor **Sub New** solo se ejecutará una vez, cuando se cree un objeto.

Ejemplo de Sub New

El siguiente ejemplo muestra cómo utilizar el constructor **Sub New**:

```
Public Sub New( )
    ' Perform simple inicialización
    intValue = 1
End Sub
```

La siguiente línea de código crea un objeto desde una clase denominada **BankAccount**, ya definida en la aplicación.

```
Dim myAccount As New BankAccount( )
```

Podemos sobrecargar **New** y crear tantos constructores de clases como sean necesarios. Esto resulta útil si deseamos inicializar nuestro objeto cuando lo creamos.

Sobrecarga de constructores

Podemos sobrecargar constructores del mismo modo que se sobrecarga cualquier otro método de una clase. No obstante, no podemos utilizar la palabra clave **Overloads** cuando sobrecargamos constructores. El siguiente ejemplo muestra cómo sobrecargar **New** y crear múltiples constructores de clase:

```
Class BankAccount
    Private balance as Double

    Sub New( )
        ' Initialize balance
        balance = 0.0
    End Sub

    Sub New(ByVal amount As Double)
        balance = amount
    End Sub
End Class
```

Uno de los constructores del ejemplo anterior toma un parámetro. Si estamos creando un objeto desde tal clase, podemos incluir sus parámetros en la declaración. El siguiente ejemplo muestra cómo invocar el método **New** que toma un parámetro.

```
Dim myAccount As New BankAccount(120.00)
```

Cómo utilizar los destructores

- Se utilizan para eliminar recursos
- Se invocan por el entorno de ejecución antes de destruir el objeto
 - Importante: es posible que la destrucción no se produzca de modo inmediato

```
Protected Overrides Sub Finalize( )
    ' Can close connections or other resources
    conn.Close
End Sub
```

Introducción

En Visual Basic .NET, podemos controlar qué ocurre durante la destrucción de objetos utilizando procedimientos denominados *destructores*.

Finalize y Dispose

El sistema invoca al destructor **Finalize** antes de liberar el objeto. Puede utilizarse para limpiar recursos abiertos, como conexiones a bases de datos, o para liberar otros recursos. Sin embargo, existe una demora entre el momento en que un objeto pierde su alcance y el momento en que se invoca al destructor **Finalize**.

La ejecución de **Sub Finalize** provoca una ligera pérdida en el rendimiento, por ello únicamente debería definirse un método **Sub Finalize** cuando sea necesario liberar objetos explícitamente.

Nota Visual Basic .NET permite un segundo tipo de destructor, denominado **Dispose**, que puede ser invocado explícitamente en cualquier momento para liberar recursos de forma inmediata. **Dispose** no se incluye en el ámbito de este curso. Si desea más información sobre **Dispose**, puede consultar “Duración de los objetos: cómo se crean y destruyen” en la documentación de Visual Studio .NET.

Ejemplo de Sub Finalize

El siguiente ejemplo muestra cómo utilizar el destructor **Finalize**:

```
Protected Overrides Sub Finalize( )
    ' Can close connections or other resources
    conn.Close
End Sub
```

Protected es un modificador de acceso que establece el nivel de accesibilidad.

Lección: Uso de miembros compartidos

- Cómo utilizar miembros de datos compartidos
- Cómo utilizar métodos compartidos

Introducción Esta lección describe cómo utilizar miembros de datos compartidos y métodos compartidos. Los miembros compartidos pueden utilizarse para contadores o para cualquier dato común o métodos habituales requeridos por todas las instancias de una clase.

Estructura de la lección Esta lección incluye los siguientes tópicos y actividades:

- Cómo utilizar los miembros de datos compartidos
- Cómo utilizar los métodos compartidos
- Práctica: creación de métodos compartidos

Objetivos de la lección En esta lección, aprenderemos a:

- Utilizar los miembros de datos compartidos para compartir datos entre instancias de clases.
- Utilizar los métodos compartidos.

Cómo utilizar miembros de datos compartidos

- Los miembros de datos compartidos permiten que múltiples instancias hagan referencia a una única variable de nivel de clase

```
Class SavingsAccount
    Public Shared InterestRate As Double
    Public Name As String, Balance As Double
    . . .
End Class
```

```
SavingsAccount.InterestRate = 0.03
```

Introducción	En Visual Basic .NET, los miembros de datos compartidos pueden utilizarse para permitir que múltiples instancias de una clase hagan referencia a una única variable a nivel de clase.
Sintaxis	Utilizaremos la siguiente sintaxis para declarar miembros de datos compartidos:
	<i>NiveldeAcceso Shared MiembrodeDatos As TipodeDatos</i>
Niveles de acceso	Los miembros de datos compartidos están directamente enlazados a la clase, y podemos declararlos como públicos o privados. Si declaramos los miembros de datos como públicos, estarán accesibles para cualquier código que pueda acceder a la clase. Si declaramos los miembros de datos como privados, proporcionaremos propiedades compartidas públicas para acceder a la propiedad compartida privada.
Ejemplo	El siguiente ejemplo muestra cómo crear una clase de cuenta de ahorro (<i>SavingsAccount</i>) que utilice un miembro de datos compartido público para mantener los tipos de interés para una cuenta de ahorro:

```
Class SavingsAccount
    Public Shared InterestRate As Double

    Public Function CalculateInterest( ) As Double
        ...
    End Function
End Class
```

El valor del miembro de datos *InterestRate* de la clase **SavingsAccount** puede establecerse globalmente con independencia del número de instancias de la clase que estén en uso. El valor se utiliza para calcular el interés del saldo actual.

Invocar miembros de datos compartidos desde un cliente

Después de crear una clase que utilice miembros de datos compartidos públicos, podemos invocar los miembros de datos de esa clase desde una aplicación cliente. El siguiente código muestra cómo invocar la clase **SavingsAccount** y sus miembros de datos desde una aplicación cliente:

```
Sub Test( )
    SavingsAccount.InterestRate = 0.03

    Dim myAccount As New SavingsAccount( )
    Dim yourAccount As New SavingsAccount( )

    MessageBox.Show(myAccount.CalculateInterest( ))
    MessageBox.Show(yourAccount.CalculateInterest( ))
End Sub
```

En el examen de este código, se observa lo siguiente:

- *InterestRate* puede establecerse antes y después de la creación de cualquier instancia de la clase **SavingsAccount**.
- Cualquier modificación a *InterestRate* se aplicará a todas las instancias de la clase **SavingsAccount**.

Propiedades compartidas

También pueden crearse propiedades compartidas en las clases. El siguiente ejemplo muestra cómo declarar una propiedad compartida denominada **Rate** en la clase **SavingsAccount**:

```
Class SavingsAccount
    Private Shared interestRate As Double

    Shared Property Rate( ) As Double
        Get
            Return interestRate
        End Get
        Set(ByVal Value As Double)
            interestRate = Value
        End Set
    End Property
End Class
```

Invocar propiedades compartidas desde un cliente

Una vez declarada la propiedad compartida **Rate**, puede utilizarse en una aplicación cliente en lugar de acceder directamente al miembro de datos compartido *interestRate*. Podemos invocar una propiedad compartida calificándola con el nombre de clase o con el nombre de variable de una instancia específica de la clase.

El siguiente código muestra cómo invocar una propiedad compartida calificándola con el nombre de la clase:

```
SavingsAccount.Rate = 0.03
```

El siguiente código muestra cómo invocar una propiedad compartida utilizando el nombre de variable de una instancia específica de la clase:

```
myAccount.Rate = 0.04
```

Cómo utilizar los métodos compartidos

- Pueden utilizarse sin declarar una instancia de una clase
- Únicamente pueden acceder a datos compartidos

```
' TestClass code
Public Shared Function GetComputerName( ) As String
    ...
End Function
```

```
' Client code
MessageBox.Show(TestClass.GetComputerName( ))
```

Introducción

Podemos utilizar los miembros de procedimiento compartidos para diseñar funciones que pueden ser invocadas sin crear una instancia de la clase. Los procedimientos compartidos son métodos de una clase que no están asociados a una instancia específica de una clase. Los miembros de procedimiento compartidos únicamente pueden acceder a datos marcados con la palabra clave **Shared**. Por ejemplo, un método compartido no puede hacer referencia a un miembro de la instancia de una clase.

Ejemplo del uso de un miembro de procedimiento compartido

El siguiente ejemplo muestra cómo una función utilizada habitualmente, como **GetComputerName**, puede crearse como un miembro de procedimiento compartido de forma que una aplicación cliente puede utilizarla fácilmente. El cliente únicamente necesita hacer referencia al método prefijado por el nombre de la clase, ya que no se requiere ninguna instancia de la clase.

```
' TestClass code
Public Shared Function GetComputerName( ) As String
    ...
End Function

' Client code
MessageBox.Show(TestClass.GetComputerName( ))
```

Observaremos también que, en el código anterior, **Show** es un método compartido de la clase **MessageBox**.

Práctica: Creación de métodos compartidos



- En esta práctica,
 - Crearemos una clase
 - Agregaremos métodos compartidos
 - Utilizaremos métodos compartidos

En esta práctica, crearemos y utilizaremos una clase que tiene un método compartido para convertir grados Fahrenheit en grados Celsius.

⚡ Crear una clase nueva

1. Abrir una nueva aplicación Windows en Visual Basic .NET. Establecer **TemperatureApplication** como nombre del proyecto y ubicarlo en una carpeta llamada SharedMethod.
2. Crear una nueva clase y denominarla **TemperatureConverter**.
3. Agregar a la nueva clase un método compartido denominado **FahrenheitToCelsius**, como sigue:

```
Public Shared Function FahrenheitToCelsius _
    (ByVal degreesFahrenheit As Double) As Double

End Function
```

4. Utilizar la siguiente formula en la función para calcular la temperatura en grados Celsius. Devolver el resultado de la función.

```
(degreesFahrenheit - 32) * 5 / 9
```

El código debería ser como el siguiente:

```
Public Shared Function FahrenheitToCelsius _
    (ByVal degreesFahrenheit As Double) As Double

    Return (degreesFahrenheit - 32) * 5 / 9

End Function
```

⚡ Invocar métodos de la clase

1. Abrir Form1 en el proyecto.
2. Agregar un botón al formulario.
3. Modificar el evento **Click** del botón para invocar el método compartido, como se muestra en las siguientes líneas de código:

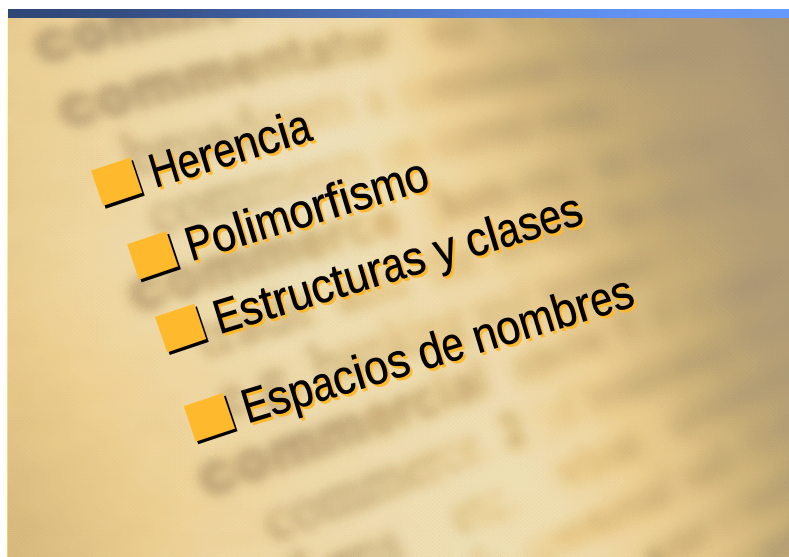
```
MessageBox.Show(TemperatureConverter.FahrenheitToCelsius _  
                (212))
```

4. Ejecutar el programa y comprobar el resultado.
5. Cerrar la aplicación.

Archivos de solución

Los archivos de solución de esta práctica se encuentran en la carpeta SharedMethod\Solution dentro del fichero practs10.zip.

Lección: herencia, polimorfismo y espacios de nombres



Introducción Podemos agregar funcionalidades a las clases en el .NET Framework o a nuestras propias clases utilizando herencia y polimorfismo. Esta lección describe la herencia y el polimorfismo. También compara clases y estructuras.

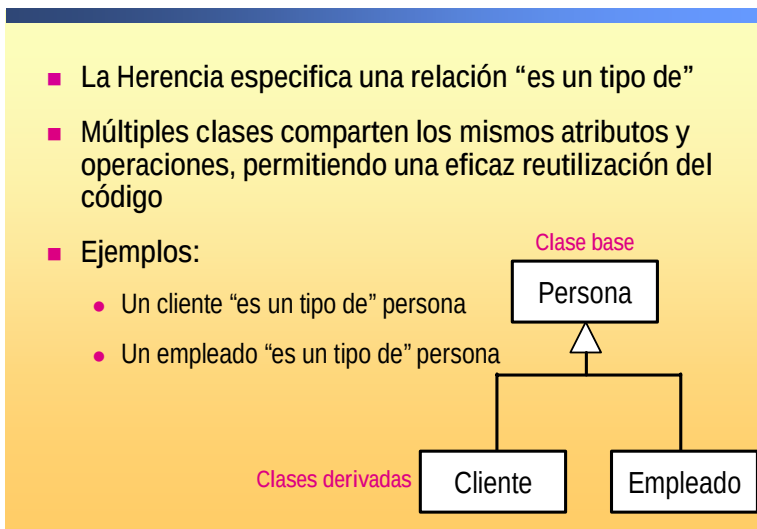
Estructura de la lección Esta lección incluye los siguientes tópicos y actividades:

- ¿Qué es la herencia?
- Cómo heredar de una clase
- ¿Qué es el polimorfismo?
- Comparación de clases y estructuras
- Cómo organizar clases en espacios de nombres

Objetivos de la lección En esta lección, aprenderemos a:

- Explicar el concepto de herencia
- Explicar el concepto de polimorfismo
- Comparar clases y estructuras
- Explicar los espacios de nombres

¿Qué es la herencia?



Introducción

En la programación orientada a objetos, podemos compartir las características de una clase base en otras clases derivadas de la clase base. Esto se denomina herencia.

Definición

Herencia es el concepto de reutilizar atributos y operaciones comunes de una clase base en una clase derivada.

Ejemplo de herencia

Imaginemos tres clases: **Cliente**, **Empleado** y **Persona**. Los atributos y operaciones de la clase base **Persona** también pueden aplicarse a **Cliente** o **Empleado**. La reutilización de estos atributos y operaciones es una técnica eficaz.

Visual Basic .NET soporta herencia simple a nivel de clase. Es decir, una clase únicamente puede heredar de una sola clase base. Esto se muestra en el ejemplo de la ilustración anterior. Otros lenguajes, como C++, permiten que una clase herede de múltiples clases.

Cómo heredar de una clase

- Una clase derivada hereda de una clase base
- Pueden heredarse propiedades, métodos, miembros de datos, eventos y controladores de eventos (dependiendo del ámbito)
- Palabras clave
 - **Inherits**: hereda de una clase base
 - **NotInheritable**: no es heredable
 - **MustInherit**: no pueden crearse instancias de la clase; deben heredarse como una clase base

Introducción	En Visual Basic .NET, la herencia puede utilizarse para derivar una clase de una clase existente. La clase derivada puede heredar todas las propiedades, métodos, miembros de datos, eventos y controladores de eventos de la clase base, facilitando la reutilización de la clase base por toda la aplicación.
La palabra clave Inherits	La palabra clave Inherits se utiliza para definir una clase derivada que heredará de una clase base existente.
Ejemplo	El siguiente ejemplo muestra cómo utilizar la palabra clave Inherits :

```
Public Class CheckingAccount
    Inherits BankAccount
    Private Sub ProcessCheck( )
        ' Add code to process a check drawn on this account
    End Sub
End Class
```

Nota Podemos utilizar la palabra clave **MyBase** para invocar métodos en una clase base cuando invalidan métodos en una clase derivada. También podemos utilizar la palabra clave **MyBase** para invocar el constructor y el destructor de la clase base en nuestra clase derivada.

La palabra clave NotInheritable

La palabra clave **NotInheritable** se utiliza para definir una clase que no puede utilizarse como clase base para herencia. Si otra clase intenta heredar de esta clase, se generará un error de compilación.

Ejemplo

El siguiente ejemplo muestra cómo utilizar la palabra clave **NotInheritable**:

```
Public NotInheritable Class TestClass
...
End Class
Public Class DerivedClass
' The following line generates a compiler error
    Inherits TestClass
...
End Class
```

La palabra clave MustInherit

La palabra clave **MustInherit** se utiliza para definir clases que no están pensados para ser utilizados directamente como objetos instanciados. La clase resultante debe ser heredada como una clase base para utilizarla en el objeto de una clase derivada instanciada.

Ejemplo

El siguiente ejemplo muestra cómo utilizar la palabra clave **MustInherit**:

```
Public MustInherit Class BaseClass
...
End Class
...
```

Si el código cliente intenta crear una instancia de este tipo de clase, se generará un error de compilación, como se muestra en el siguiente ejemplo:

```
' Client code
' The following line generates a compiler error
Dim x As New BaseClass( )
```

La palabra clave Protected

Utilizamos el acceso **Protected** para limitar el ámbito de una propiedad, método, miembro de datos, evento o controlador de eventos a la clase que los define y cualquier clase derivada basada en esa clase base.

Ejemplo

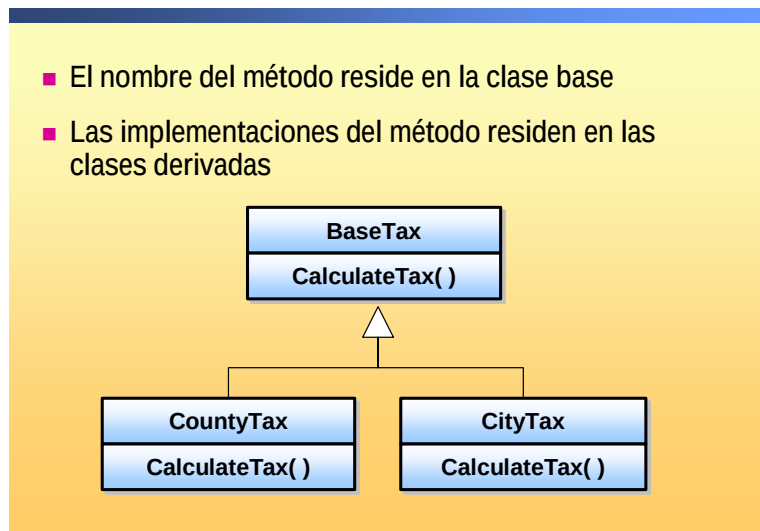
El siguiente ejemplo muestra cómo utilizar la palabra clave **Protected**:

```
Public Class BaseClass
    ' Accessible anywhere
    Public counter As Integer

    ' Accessible only in this class or a derived class
    Protected name As String
...
End Class
```

¿Qué es el polimorfismo?

- El nombre del método reside en la clase base
- Las implementaciones del método residen en las clases derivadas



Introducción

La mayoría de sistemas de programación orientada a objetos proporcionan polimorfismo mediante herencia. El polimorfismo basado en herencia implica la definición de métodos en una clase base y sobrecargarlos con nuevas implementaciones en clases derivadas.

Definición

Polimorfismo hace referencia a la capacidad de definir múltiples clases con diferentes funcionalidades pero con métodos o propiedades de nombres idénticos que pueden utilizarse de forma intercambiable por el código cliente en tiempo de ejecución. El nombre del método reside en la clase base. Las implementaciones de métodos residen en las clases derivadas. Para gestionar esto, únicamente puede declararse en la clase base el nombre del método (no el código que proporciona la funcionalidad del método).

Ejemplo de polimorfismo

Supongamos que definimos una clase denominada **BaseTax** que proporciona funcionalidad básica para calcular el impuesto sobre las ventas de un estado. Las clases derivadas de **BaseTax**, como **CountyTax** o **CityTax**, podrían implementar métodos como **CalculateTax**.

Polimorfismo hace referencia al hecho de que la implementación del método **CalculateTax** podría ser distinta en cada una de las clases derivadas. Por ejemplo, el tipo impositivo de un condado (*county tax rate*) podría ser distinto del tipo impositivo de una ciudad (*city tax rate*). Las clases que hereden de **BaseTax** tendrán un método **CalculateTax**, pero el modo como se calcule realmente el impuesto podría variar en cada una de las clases derivadas.

Comparación entre clases y estructuras

Clases	Estructuras
Pueden definir miembros de datos, propiedades y métodos	Pueden definir miembros de datos, propiedades y métodos
Soportan constructores e inicialización de miembros	No constructor predeterminado ni inicialización de miembros
Soportan el método Finalize	No soportan el método Finalize
Extensibles por herencia	No soportan herencia
Tipo Reference	Tipo Value

Introducción

Clases y estructuras son similares en varios aspectos: ambas pueden definir miembros de datos, propiedades y métodos. Sin embargo, las clases proporcionan algunas características avanzadas que podemos utilizar.

La siguiente tabla compara clases y estructuras:

	Clases	Estructuras
Inicialización	Soportan constructores y la inicialización de miembros.	No tienen constructor predeterminado ni inicialización de miembros.
Método Finalize	Soportan el método Finalize .	No soportan el método Finalize . Los recursos deben liberarse manualmente.
Herencia	Extensible mediante herencia.	No soportan herencia.
Tipo de datos	Tipo de datos Referencia. Cuando una variable de objeto se pasa a una función, se pasa la referencia a la dirección de los datos en lugar de los propios datos. Asignar una variable de clase a otra apunta ambas variables al mismo objeto. Por tanto, las actualizaciones de una de las variables afectarán a la otra.	Tipo de datos Valor. Cuando una variable de estructura se pasa a una función, los datos reales deben copiarse a la función. Asignar una variable de estructura a otra crea una copia real de la estructura. Por tanto, las actualizaciones de una de las variables no afectarán a la otra. La diferencia en tipo de datos tiene un efecto significativo en el rendimiento de la aplicación. Una clase con gran cantidad de datos internos tendrá un mejor rendimiento que una estructura de datos de gran tamaño en esas condiciones.

Cómo organizar clases en espacios de nombres

- Los espacios de nombres son un sistema organizativo
- Los espacios de nombres proporcionan nombres cualificados para las clases
 - Ejemplo: `System.Windows.Forms.Button`
- Para importar un espacio de nombres:
 - En el nivel de proyecto, agregar una referencia a la DLL que contiene el espacio de nombres
 - Utilizar la palabra clave `Imports`

Introducción

Los espacios de nombres se utilizan como sistemas organizativos (una forma de presentar componentes del programa relacionados lógicamente que están disponibles para otros programas y aplicaciones). Un espacio de nombres puede contener otros espacios de nombres y tipos.

Ejemplo de un espacio de nombres

System.Windows.Forms es un ejemplo de espacio de nombres. Este espacio de nombres proporciona las clases relevantes para crear formularios. El espacio de nombres **Forms** está contenido en el espacio de nombres **Windows**, el cual está contenido en el espacio de nombres **System**.

Nombres cualificados

Para hacer referencia a una clase utilizando su nombre cualificado, prefijamos el nombre de la clase con el espacio de nombres que contiene la clase. Por ejemplo, el nombre cualificado de la clase **Button** es **System.Windows.Forms.Button**. Con el uso de nombres cualificados, podemos declarar dos clases con el mismo nombre en distintos espacios de nombres sin conflicto. De forma predeterminada, cada archivo ejecutable que creamos con Visual Basic .NET contiene un espacio de nombres con el mismo nombre que el proyecto.

Accesibilidad

Los espacios de nombres son siempre **Public**, por tanto, no podemos declarar un espacio de nombres con un modificador de acceso. Sin embargo, los componentes en el espacio de nombres pueden tener acceso **Public** o **Friend**. Si el modificador de acceso no se ha declarado, el tipo de acceso predeterminado es **Friend**.

Definir un espacio de nombres

En Visual Basic .NET, utilizamos la instrucción **Namespace** para definir un nuevo espacio de nombres, el cual encapsula las clases que creamos, como muestra el siguiente ejemplo:

```
Namespace CompVB
    Public Class StringComponent
        ...
    End Class
End Namespace
```

Uso de un espacio de nombres

A nivel de proyecto, debemos incluir una referencia a la librería de enlace dinámico (DLL) que contiene el espacio de nombres. En Visual Basic .NET, utilizamos la instrucción **Imports** para importar los tipos contenidos en el espacio de nombres para que puedan ser referenciados directamente. El siguiente código muestra el uso de la instrucción **Imports**:

```
Imports System.Windows.Forms

Public Class Form1
    Inherits Form
```

Si omitimos la instrucción **Imports**, deberemos utilizar el nombre cualificado del espacio de nombres, como muestra el siguiente ejemplo:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
```