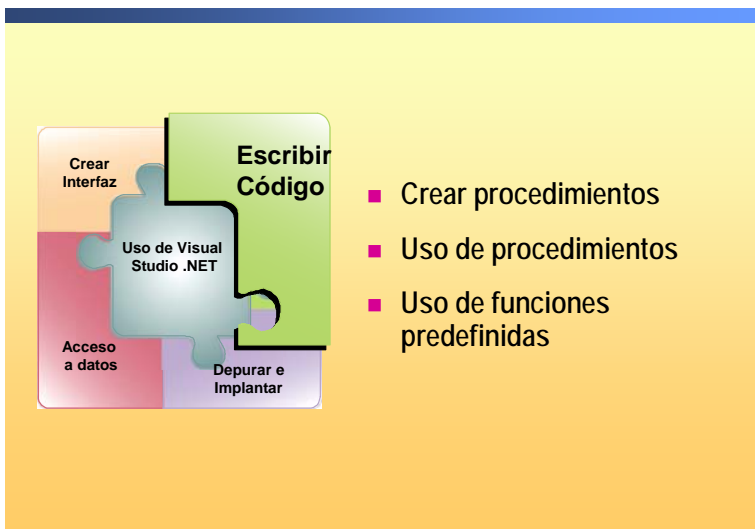


Funciones. Subrutinas y procedimientos

Índice

| | |
|--|----|
| Descripción | 1 |
| Lección: Crear procedimientos | 2 |
| Lección: Uso de procedimientos | 17 |
| Lección: Uso de funciones predefinidas | 33 |

Descripción



Introducción

El desarrollo de una aplicación, especialmente si se trata de un proyecto de gran tamaño, es más fácil si se divide en piezas más pequeñas. El uso de procedimientos puede ayudarnos a agrupar nuestro código en secciones lógicas y condensar tareas repetidas o compartidas, como cálculos utilizados frecuentemente. En este módulo, aprenderemos a crear y utilizar procedimientos.

Objetivos

En este módulo, aprenderemos a:

- Diferenciar entre un procedimiento **Sub** y un procedimiento **Function**.
- Crear e invocar procedimientos **Sub** y **Function**.
- Escribir procedimientos en módulos para permitir la reutilización del código.
- Pasar argumentos por valor y por referencia.
- Utilizar funciones predefinidas en el código de nuestra aplicación.

Lección: Crear procedimientos

- ¿Qué son los procedimientos?
- Cómo crear procedimientos **Sub**
- Cómo crear procedimientos **Function**
- Cómo declarar argumentos en procedimientos
- Cómo utilizar argumentos opcionales
- Reutilización del código

Introducción Esta lección explica cómo crear procedimientos **Sub** y **Function**, cómo declarar argumentos en un procedimiento y cómo crear procedimientos en un módulo.

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

- ¿Qué son los procedimientos?
- Cómo crear procedimientos **Sub**
- Cómo crear procedimientos **Function**
- Cómo declarar argumentos en procedimientos
- Multimedia: pasar argumentos
- Cómo utilizar argumentos opcionales
- Multimedia: escribir código reutilizable
- Reutilización de código
- Práctica: crear una Función en un Módulo

Objetivos de la lección En esta lección, aprenderá a:

- Describir y crear un procedimiento **Sub**.
- Describir y crear un procedimiento **Function**.
- Explicar la diferencia entre pasar argumentos por valor y pasar argumentos por referencia a un procedimiento.
- Declarar argumentos, incluyendo los opcionales, en un procedimiento.
- Crear procedimientos en un módulo.

¿Qué son los procedimientos?

- Los procedimientos son las sentencias de código ejecutable de un programa, encerrados por una sentencia de declaración y una sentencia **End**
- Tres tipos:
 - Procedimientos **Sub** (incluyendo procedimientos **Sub** de eventos)
 - Procedimientos **Function**
 - Procedimientos **Property**
- Permitir la reutilización de código
- Declarados como **public** de forma predeterminada

Definición

Los procedimientos son las sentencias de código ejecutable de un programa. Las instrucciones de un procedimiento están delimitadas por una instrucción de declaración y una instrucción **End**.

Nota Es posible que encontremos los términos *métodos*, *procedimientos* y *funciones* de forma intercambiable en varias referencias. Este módulo sigue la terminología de la documentación de Microsoft® Visual Studio® .NET.

Tipos de procedimientos

Existen tres tipos de procedimientos en Microsoft Visual Basic® .NET: procedimientos **Sub**, procedimientos **Function** y procedimientos **Property**.

- Los procedimientos **Sub** realizan acciones pero no devuelven un valor al procedimiento que origina la llamada. Los controladores de eventos son procedimientos **Sub** que se ejecutan en respuesta a un evento.
- Los procedimientos **Function** pueden devolver un valor al procedimiento que origina la llamada. La instrucción **MessageBox.Show** es un ejemplo de función.
- Los procedimientos **Property** devuelven y asignan valores de propiedades de clases, estructuras o módulos.

Nota Si se desea más información sobre la creación y uso de procedimientos **Property**, consultar el Módulo 7.

Uso de procedimientos

Un procedimiento puede ser invocado, o *llamado*, desde otro procedimiento. Cuando un procedimiento llama a otro procedimiento, se transfiere el control al segundo procedimiento. Cuando finaliza la ejecución del código del segundo procedimiento, éste devuelve el control al procedimiento que lo invocó.

Debido a esta funcionalidad, los procedimientos resultan útiles para realizar tareas repetidas o compartidas. En lugar de escribir el mismo código más de una vez, podemos escribir un procedimiento e invocarlo desde varios puntos de nuestra aplicación o desde otras aplicaciones.

Accesibilidad del procedimiento

Utilizamos un modificador de acceso para definir la *accesibilidad* de los procedimientos que escribimos (es decir, el permiso para que otro código invoque al procedimiento). Si no especificamos un modificador de acceso, los procedimientos son declarados *public* de forma predeterminada.

La siguiente tabla muestra las opciones de accesibilidad para declarar un procedimiento dentro de un módulo:

| Modificador de acceso | Descripción |
|-----------------------|---|
| Public | Ninguna restricción de acceso |
| Friend | Accesible desde el programa que contiene la declaración y desde cualquier otro lugar del mismo ensamblado |
| Private | Accesible únicamente en el módulo que contiene la declaración |

Nota El modificador de acceso **Protected** únicamente puede utilizarse en procedimientos declarados dentro de una clase. Si se desea más información sobre la declaración de procedimientos en una clase, consultar el Módulo 7.

Cómo crear procedimientos Sub

Los procedimientos Sub realizan acciones pero no devuelven un valor al procedimiento que realiza la llamada

```
[accessibility] Sub subname[(argumentlist)]
    ' Sub procedimiento statements
End Sub
```

Ejemplo:

```
Private Sub AboutHelp( )
    MessageBox.Show("MyProgram V1.0", "MyProgram Help")
End Sub
```

Definición

Un procedimiento **Sub** es una serie de instrucciones de Visual Basic delimitadas por las instrucciones **Sub** y **End Sub**. Los procedimientos **Sub** realizan acciones pero no devuelven un valor al procedimiento que origina la llamada.

Sintaxis para crear un procedimiento Sub

Utilicemos la siguiente sintaxis para crear un procedimiento **Sub**:

```
[accessibility] Sub subname[(argumentlist)]
    ' Statements of the Sub procedure go here
End Sub
```

Ejemplo de procedimiento Sub

El siguiente código crea un procedimiento **Sub** (**Sub AboutHelp**) que utiliza un cuadro de mensaje para mostrar un nombre de producto y un número de versión:

```
Private Sub AboutHelp( )
    MessageBox.Show("MyProgram V1.0", "MyProgram Help")
End Sub
```

Cómo crear procedimientos Function

Los procedimientos Function realizan acciones y pueden devolver un valor al programa que realiza la llamada

```
[accessibility] Function name[(argumentlist)] As datatype
' Function statements, including optional Return
' statement
End Function
```

Ejemplo:

```
Public Function DoubleTheValue(ByVal J As Double) As _
    Double
    . . .
    Return J*2
    . . .
End Function
```

Definición

Un procedimiento **Function** es una serie de instrucciones Visual Basic delimitadas por las instrucciones **Function** y **End Function**. Los procedimientos **Function** son similares a los procedimientos **Sub**, pero las funciones pueden devolver un valor al programa que origina la llamada.

Sintaxis para crear a Function procedimiento

Utilicemos la siguiente sintaxis para crear un procedimiento **Function**:

```
[accessibility] Function functionname[(argumentlist)] As _
    datatype
' Statements of the function go here, including optional
' Return statement
End Function
```

Ejemplo de un procedimiento Function

El siguiente código crea una función denominada **Square** que devuelve el cuadrado de un número entero (*integer*):

```
Function Square(SquareValue As Integer) As Integer
    Square = SquareValue * SquareValue
End Function
```

Valores de retorno

El valor que devuelve un procedimiento **Function** al programa que origina la llamada se denomina *valor de retorno*. La función devuelve el valor en una de las dos formas siguientes:

- Asigna un valor al nombre de su propia función en una o más instrucciones dentro del procedimiento. El control no se devuelve al programa que origina la llamada hasta que se ejecuta una instrucción **Exit Function** o **End Function**.
La instrucción **Exit Function** provoca la salida inmediata de un procedimiento **Function**. Cualquier número de instrucciones **Exit Function** pueden aparecer en cualquier lugar del procedimiento.
- Utiliza una instrucción **Return** para especificar el valor devuelto, y devuelve el control inmediatamente al programa que origina la llamada.

La ventaja de asignar el valor devuelto al nombre de función es que el control no se devuelve desde la función hasta que el programa encuentra una instrucción **Exit Function** o **End Function**. Esto permite asignar un valor preliminar y ajustarlo más tarde si es necesario.

Ejemplo de asignación del valor de retorno

El siguiente ejemplo asigna el valor de retorno al nombre de función **DoubleTheValue** y utiliza la instrucción **Exit Function** para volver al procedimiento de llamada:

```
Function DoubleTheValue(ByVal j As Integer) As Double
    . . .
    DoubleTheValue = j*2
    ' Control remains within the function
    . . .
    Exit Function
    ' Control returns to the calling function
    . . .
End Function
```

Si salimos de la función sin asignar un valor devuelto, la función devuelve el valor predeterminado apropiado para el tipo de datos de la función. Por ejemplo, devuelve 0 para **Byte**, **Char**, **Decimal**, **Double**, **Integer**, **Long**, **Short** y **Single**.

Ejemplo de uso de la instrucción Return

La instrucción **Return** asigna simultáneamente el valor devuelto y sale de la función, como se muestra en el siguiente ejemplo:

```
Function DoubleTheValue(ByVal j As Integer) As Double
    . . .
    Return j*2
    ' Control is immediately returned to the calling function
    . . .
End Function
```

Cómo declarar argumentos en procedimientos

- Los *argumentos* son datos pasados a procedimientos
- Podemos pasar argumentos *ByVal* o *ByRef*
 - **ByVal**: El procedimiento no puede modificar el valor de la variable original
 - **ByRef**: El procedimiento puede modificar el valor de la variable original
 - **Excepción**: Los elementos no variables no se modifican en el código que llama, aunque sean pasados por referencia
- **ByVal** es el valor predeterminado en Visual Basic .NET
- **Sintaxis y ejemplo:**

```
([ByVal|ByRef] argumentname As datatype)
```

```
(ByVal Name As String)
```

Introducción

Un procedimiento que realiza tareas repetidas o compartidas utiliza distinta información en cada llamada. Esta información puede estar formada por variables, constantes y expresiones que se pasan al procedimiento por el procedimiento que origina la llamada. Cada valor que se pasa a un procedimiento se denomina *argumento*.

Parámetros vs. argumentos

Cuando definimos un procedimiento en Visual Basic .NET, describimos los datos y los tipos de datos para los que el procedimiento está diseñado para aceptar desde un procedimiento de llamada. Los elementos definidos en el procedimiento se denominan *parámetros*.

Cuando invocamos el procedimiento, sustituimos un valor actual de cada parámetro. Los valores que asignamos en lugar de los parámetros se denominan *argumentos*.

Nota A pesar de esta sutil diferencia, los términos *argumento* y *parámetro* a menudo se utilizan indistintamente. Este módulo utiliza la terminología utilizada en la documentación de Visual Studio .NET.

Paso ByVal y ByRef

Cuando definimos un procedimiento, definimos el modo en el que otros procedimientos pueden pasar argumentos al procedimiento. Podemos escoger pasarle argumentos por referencia (**ByRef**) o por valor (**ByVal**). En Visual Basic .NET, el mecanismo predeterminado de paso de parámetros es por valor. Si no especificamos **ByVal** ni **ByRef** en nuestras definiciones de parámetros, **ByVal** se añade automáticamente a la definición del parámetro.

| Mecanismo de paso | Explicación | Implicaciones | Ventaja |
|--|---|---|--|
| Por valor Palabra clave: ByVal | El procedimiento invocado recibe una copia de los datos cuando es invocado. | Si el procedimiento invocado modifica la copia, el valor original de la variable permanece intacto. Cuando la ejecución retorna al procedimiento de llamada, la variable contiene el mismo valor que tenía antes de que el valor se pasara. | Protege la variable de ser cambiada por el procedimiento invocado. |
| Por referencia Palabra clave: ByRef | El procedimiento invocado recibe una referencia a los datos originales (la dirección de los datos en memoria) cuando es invocado. | El procedimiento invocado puede modificar la variable directamente. Cuando la ejecución retorna al procedimiento de llamada, la variable contiene el valor modificado. | El procedimiento invocado puede utilizar el argumento para devolver un nuevo valor al código de llamada. |

Excepciones

El elemento de programación que subyace en un argumento puede ser un elemento variable, cuyo valor puede ser cambiado, o un elemento no variable. Los argumentos no variables nunca son modificados en el código de llamada, aunque se pasen por referencia. El procedimiento invocado podría modificar su copia de ese argumento, pero la modificación no afectaría al elemento subyacente en el código de llamada.

La siguiente tabla muestra elementos variables y no variables.

| Elementos variables (pueden modificarse) | Elementos no variables |
|---|------------------------|
| Variables declaradas, incluyendo variables de objetos | Constantes |
| Campos (de clases) | Literales |
| Elementos de matrices | Enumeraciones |
| Elementos de estructuras | Expresiones |

Declarar argumentos

Utilizamos la misma sintaxis para declarar los argumentos para procedimientos **Sub** y procedimientos **Function**. Declaramos cada argumento de un procedimiento del mismo modo en que declaramos una variable, especificando el nombre del argumento y el tipo de datos. También podemos especificar el mecanismo de paso y si el argumento es opcional.

La sintaxis para cada argumento en la lista de argumentos de un procedimiento es como sigue:

```
([ByVal|ByRef] [ParamArray] nombreargumento As datatype)
```

Nota Si se desea más información sobre la sintaxis **ParamArray**, leer *Cómo pasar matrices a procedimientos* en este módulo.

Si el argumento es opcional, debemos incluir también la palabra clave **Opcional** y proporcionar un valor predeterminado en la declaración, como sigue:

```
Opcional [ByVal|ByRef] nombreargumento As datatype =  
defaultvalue
```

Nota Si deseamos más información sobre argumentos opcionales, leer *Cómo utilizar argumentos opcionales* en este módulo.

Ejemplo de declaración de un argumento

En el siguiente ejemplo, el procedimiento **Sub Hello** está diseñado para tomar un argumento **Name** de tipo **String** por valor desde un procedimiento de llamada.

```
Public Sub Hello(ByVal Name As String)  
    MessageBox.Show("Hello, " & Name & "!!")  
End Sub
```

Nota No es necesario que los nombres de argumentos utilizados cuando se invoca un procedimiento coincidan con los nombres de parámetros utilizados para definir el procedimiento.

Cómo utilizar argumentos opcionales

■ Reglas para declarar argumentos opcionales:

- Especificar un valor predeterminado
- El valor predeterminado debe ser una expresión constante
- Los argumentos que sigan a un argumento opcional también deben ser opcionales

■ Sintaxis:

```
(Optional [ByVal|ByRef] argumentname As datatype = defaultvalue)
```

■ Ejemplo:

```
Function Add (ByVal value1 As Integer, ByVal value2 As _  
Integer, Optional ByVal value3 As Integer = 0) As Integer
```

Introducción

Podemos especificar que el argumento de un procedimiento es opcional y no es necesario proporcionarlo cuando el procedimiento es invocado. Esto ofrece flexibilidad cuando nuestro procedimiento es invocado por otro procedimiento. El usuario puede decidir proporcionar o no un argumento.

Declarar un argumento opcional

Los argumentos opcionales están indicados por la palabra clave **Optional** en la definición del procedimiento. Además, cuando declaramos un argumento opcional, se aplican las siguientes reglas:

- Debe especificarse un valor predeterminado para todos los argumentos opcionales.
- El valor predeterminado de un argumento opcional debe ser una expresión constante.
- Todos los argumentos que sigan a un argumento opcional en la definición del procedimiento también deben ser opcionales.

El siguiente código muestra la sintaxis para declarar un argumento opcional:

```
Optional [ByVal|ByRef] nombreargumento As datatype =  
defaultvalue
```

Ejemplo de argumento opcional

El siguiente ejemplo muestra una declaración de procedimiento con un argumento opcional:

```
Function Add(ByVal value1 As Integer, ByVal value2 As _  
Integer, Optional ByVal value3 As Integer = 0) As Integer  
' The default valor for the optional argument is 0
```

Ejemplo de argumento opcional incorrecto

El siguiente ejemplo contiene un error; recordemos que los argumentos que siguen a un argumento opcional también deben ser opcionales.

```
Function Add(ByVal value1 As Integer, Optional ByVal _  
    value2 As Integer = 0, ByVal value3 As Integer) As Integer  
    ' Causes an error
```

Procedimientos de llamada con argumentos opcionales

Cuando invocamos un procedimiento con un argumento opcional, podemos escoger entre proporcionar o no el argumento. Si no proporcionamos el argumento, el procedimiento utiliza el valor predeterminado declarado para ese argumento.

Cuando omitimos uno o más argumentos opcionales en la lista de argumentos, utilizamos comas sucesivas para separar los espacios marcando sus posiciones. La siguiente invocación proporciona los argumentos primero y cuarto, pero no proporciona el segundo ni el tercero:

```
SubCount(arg1, , , arg4)  
' Leaves out arg2 and arg3
```

Reutilización de código

| Use un... | Para... | Ejemplos |
|-------------------|---|------------------------------|
| Estructura | Objetos que no necesitan ser extendidos | Size Point |
| Módulo | Funciones de utilidad y datos globales | Conversión de temperatura |
| Clase | Extende objetos u objetos que necesitan cleanup | Formularios Botones |

■ Crear un módulo:

```
[Public|Friend] Module ModuleName
    .
    .
    .
End Module
```

Introducción

Uno de los procesos más importantes en la creación de una aplicación basada en Visual Basic es diseñar código para su reutilización. El modo como escribimos el código afecta a su reutilización.

Escribir código para reutilizar

Podemos escribir código para ser reutilizado, incluyendo procedimientos, en estructuras, módulos o clases. La siguiente tabla proporciona una descripción de las situaciones en las que deberíamos escoger cada una de estas opciones:

| Usar un(a)... | para... | Ejemplo |
|---------------|--|--|
| Estructura | Crear objetos que no necesitan ser extendidos y que tienen un tamaño de instancia pequeño | Size y Point son estructuras disponibles en la biblioteca de clases del Microsoft .NET Framework |
| Módulo | Proporcionar funciones de utilidad y datos globales para su uso por múltiples módulos o clases | Funciones de utilidad como conversión de temperatura, cálculo de área, acceso a datos, etc., necesarias para múltiples módulos |
| Clase | Extender objetos, o para objetos que necesitan liberar recursos | Clase Forms , clase Button , etc. |

Escribir procedimientos en una estructura

El siguiente código muestra cómo podemos escribir un procedimiento en una estructura. Asumimos que las variables *x*, *y* y *z* del ejemplo ya han sido declaradas.

```
Structure TableDimensions
  Private legHeight, topWidth, topDepth As Integer
  Public Sub New(ByVal legHeight As Integer, _
    ByVal topWidth As Integer, ByVal topDepth as Integer)
    Me.legHeight = x
    Me.topWidth = y
    Me.topDepth = z
  End Sub
End Structure
```

Crear un módulo

Para crear un módulo, añadimos primero un módulo a nuestro proyecto. A continuación, escribimos las instrucciones del código que definen los datos y procedimientos de nuestro módulo.

🔍 Añadir un módulo a un proyecto

1. Si el Explorador de soluciones no está abierto, en el menú **Ver**, hacer clic en **Explorador de soluciones**.
2. En el Explorador de soluciones, hacer clic con el botón derecho en nuestro proyecto, seleccionar **Agregar** y, a continuación, hacer clic en **Agregar nuevo elemento**.
3. En el cuadro de diálogo **Agregar nuevo elemento**, en el cuadro **Nombre**, escribir un nombre para su módulo, seleccione **Módulo** en las **Plantillas** y, a continuación, hacer clic en **Abrir**.

Sintaxis

La siguiente sintaxis declara un bloque de módulo:

```
[Public|Friend] Module nombremódulo
  ' Add classes, properties, methods, fields, and events for
  ' the module
End Module
```

Accesibilidad de un módulo

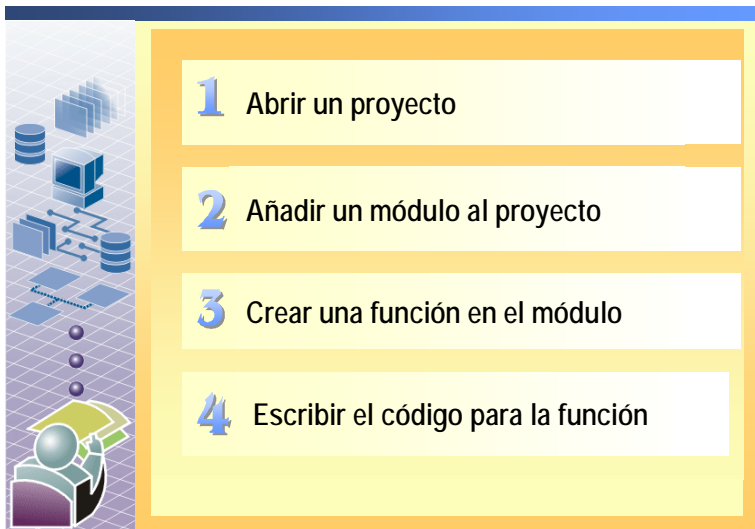
Al igual que con los procedimientos y las variables, utilizamos los modificadores de acceso para definir la accesibilidad de un módulo. Si no utilizamos un modificador de acceso, los módulos se declaran **Friend** de modo predeterminado.

La siguiente tabla define los modificadores de acceso disponibles para un módulo:

| Modificador de acceso | Definición |
|-----------------------|--|
| Public | Ninguna restricción de acceso |
| Friend | Accesible desde dentro del programa que contiene la declaración y desde cualquier lugar del mismo ensamblado |

Nota Si desea más información sobre clases, consulte el Módulo 7.

Práctica: crear una función en un módulo



En esta práctica, añadirá un módulo a un proyecto y creará una función en el módulo. La función tomará **height** y **width** como argumentos y devolverá **Area**.

Nota Utilizará el código creado en esta práctica como código de inicio para la práctica *Utilización del valor devuelto de una función*, que realizará más adelante en este módulo.

⚡ Crear una función en un módulo

1. Abra un nuevo proyecto en Visual Basic .NET. Utilice la plantilla Aplicación para Windows. Asigne al proyecto el nombre **CalculateArea** y seleccione la carpeta donde quiera crearlo. Hacer clic en **OK**.
2. Añadir un nuevo módulo al proyecto. Para ello, en el menú **Proyecto**, hacer clic en **Agregar nuevo elemento**. En el panel **Plantillas**, hacer clic en **Módulo**, mantener el nombre predeterminado, y hacer clic en **Abrir**.
3. En el módulo, cree una nueva función denominada **Area** que devuelva un tipo de datos **Single**.
 - a. La función debería tomar dos argumentos denominados **height** y **width** por valor. Declare cada argumento como **Single**.
 - b. En el cuerpo de la función, escriba el código para multiplicar **height** y **width** y para asignar el valor devuelto a **Area**, como se muestra en el siguiente código:

```
Area = height * width
```

4. Guarde su proyecto. Nuestro código completo debería ser como el siguiente:

```
Module Module1
    Function Area(ByVal height As Single, _
        ByVal width As Single) As Single
        Area = height * width
    End Function
End Module
```

Nota No podrá probar la función hasta que finalice la práctica *Utilización del valor devuelto de una función*.

Archivos de solución

Los archivos de solución se encuentran en la carpeta Area\Solution dentro del archivo practs04.zip.

Lección: uso de procedimientos

- Cómo utilizar procedimientos Sub
- Cómo utilizar procedimientos Function
- Cómo pasar matrices a procedimientos
- Cómo crear un Sub Main

Introducción

Uno de los principales beneficios del uso eficaz de procedimientos es la reutilización de código. Los procedimientos que creamos en un programa pueden utilizarse en ese programa y en otros proyectos, frecuentemente con poca o nula modificación. Los procedimientos son útiles para tareas repetidas o compartidas, como cálculos utilizados frecuentemente.

Esta lección describe cómo utilizar procedimientos **Sub** y **Function**, cómo utilizar la sintaxis **ParamArray** para pasar argumentos, y cómo modificar el inicio de la aplicación para crear un procedimiento **Sub Main**.

Estructura de la lección

Esta lección incluye los siguientes temas y actividades:

- Cómo utilizar procedimientos **Sub**
- Cómo utilizar procedimientos **Function**
- Práctica: utilización del valor devuelto de una función
- Cómo pasar matrices a procedimientos
- Cómo crear un **Sub Main**
- Práctica: crear un **Sub Main**

Objetivos de la lección


En esta lección, aprenderá a:

- Invocar y pasar argumentos a un procedimiento **Sub**.
- Pasar argumentos a una función y utilizar un valor devuelto.
- Utilizar **ParamArray** para declarar una matriz en el argumento de un procedimiento.
- Modificar el inicio de la aplicación creando un procedimiento **Sub Main**.

Cómo utilizar los procedimientos Sub

```
Public Sub Hello(ByVal name As String)
    MessageBox.Show("Hello " & name)
End Sub
```

```
Sub Test( )
    Hello("John")
End Sub
```



Introducción

Para utilizar un procedimiento **Sub**, lo invocamos desde otro procedimiento.

Flujo de código

Cada vez que se invoca un procedimiento **Sub**, se ejecutan sus instrucciones, empezando por la primera instrucción ejecutable después de la instrucción **Sub** y finalizando con la primera instrucción **End Sub**, **Exit Sub** o **Return** encontrada. Después de que el procedimiento **Sub** ejecute nuestro código, devuelve la ejecución del programa a la línea de código que sigue a la línea que invocó el procedimiento **Sub**.

Invocar un procedimiento Sub

La sintaxis para invocar un procedimiento **Sub** es la siguiente:

```
[Call] Subname [(ArgumentList)]
```

- Debemos invocar el procedimiento **Sub** en una línea por sí mismo en nuestro código (no puede invocarlo utilizando su nombre dentro de una expresión).
- La instrucción de llamada debe proporcionar valores para todos los argumentos que no son opcionales.
- Opcionalmente, podemos utilizar la instrucción **Call** para invocar un procedimiento **Sub**. El uso de la instrucción **Call** puede mejorar la legibilidad de nuestro programa.

Nota Los procedimientos **Sub** no devuelven un valor a la instrucción de llamada. Sin embargo, un procedimiento **Sub** pasar información de retorno al código de llamada modificando argumentos pasados por referencia.

Ejemplo de una invocación simple

El siguiente código muestra un procedimiento de evento que invoca un procedimiento **Sub** denominado **SetData**:

```
Sub DataButton_Click(...)
    SetData( )
End Sub
```

Ejemplo de uso de la instrucción Call

También puede utilizar el siguiente código para realiza la misma tarea:

```
Sub DataButton_Click(...)
    Call SetData( )
End Sub
```

Ejemplo de invocación simple incorrecta

La siguiente invocación contiene un error:

```
Sub DataButton_Click(...)
    MessageBox.Show(SetData( ))
End Sub

' Causes an error, because the Show method expects a String
' data type, not un procedure
```

Ejemplo de invocación con argumentos

Observe la siguiente definición para el procedimiento **Sub SetData**:

```
Public Sub SetData(ByVal cars As Integer, ByVal trucks As _
    Integer, ByVal vans As Integer)
    ' Code for SetData Procedure
End Sub
```

La sintaxis para invocar este procedimiento incluye el nombre del procedimiento y la lista de argumentos en paréntesis, como se muestra en el siguiente código:

```
Sub DataButton_Click(...)
    SetData(10, 20, 30)
End Sub
```

Ejemplo incorrecto de una llamada con argumentos

La siguiente llamada al procedimiento **SetData** definido en el ejemplo a continuación contiene un error.

```
Sub DataButton_Click( )
    SetData(10, 20)
End Sub

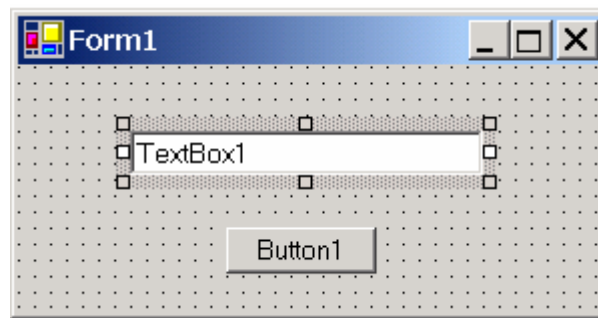
' Causes an error, because there is no valor for the third
' parametre of the SetData procedure. The calling statement
' must provide values for all arguments that are not optional.
```

Práctica (opcional) Pasar argumentos por referencia

En esta práctica, crearemos dos procedimientos. El primer procedimiento tomará un argumento por referencia. El segundo procedimiento invocará al primer procedimiento y pasará un valor por referencia. Observaremos el efecto de invocar un procedimiento y pasar un valor por referencia.

✍ Crear el interfaz de usuario

1. Abra un nuevo proyecto en Visual Basic .NET. Utilice la plantilla Aplicación para Windows. Asigne al proyecto el nombre **ByRefTest** y seleccione la carpeta donde quiera crearlo. Hacer clic en **Aceptar**.
2. Abra la vista de diseño de Form1.
3. Abra el Cuadro de herramientas. Añada un control **TextBox** y un control **Button** a su formulario. Organice los controles para que su formulario tenga un aspecto similar al de la siguiente figura:



✍ Crear un procedimiento que toma un argumento por referencia

1. Abra el Editor de código de Form1.
2. Cree un procedimiento denominado **Hello** que toma un argumento **String**, *Name*, por referencia. Nuestro código debería ser como el siguiente:

```
Public Sub Hello(ByRef Name As String)
End Sub
```

3. Para visualizar el efecto de pasar una variable por referencia cuando invoca un segundo procedimiento, añada las tres líneas de código siguientes al procedimiento **Hello**:
 - a. Mostrar el valor actual de *Name* en un cuadro de texto.
 - b. Establecer el valor de la variable *Name* a **Samantha**.
 - c. Mostrar el nuevo valor de *Name* en un cuadro de texto.

Nuestro código debería tener un aspecto similar al siguiente:

```
Public Sub Hello(ByRef Name As String)
    MessageBox.Show("Hello, " & Name & "!")
    Name = "Samantha"
    MessageBox.Show("Hello, " & Name & "!")
End Sub
```

✍ Invocar un procedimiento y pasar un argumento

- Añada un controlador de eventos para el evento **Button1_Click**. En el controlador de eventos, invoque el procedimiento **Hello**, pasando la propiedad **Text** de **TextBox1** como un argumento.

Nuestro código debería tener un aspecto similar al siguiente:

```
Private Sub Button1_Click(...) Handles Button1.Click
    Hello(TextBox1.Text)
End Sub
```

✍ Ejecutar la aplicación y probar los resultados

1. Ejecutar la aplicación.
2. Escribir nuestro nombre en **TextBox1**.
3. Hacer clic en **Button1**, comprobar si aparece el nombre y hacer clic en **OK**.
El segundo cuadro de mensaje que contiene el nombre “Samantha” se abrirá inmediatamente.
4. Antes de hacer clic en **OK**, observar el cuadro de texto. Mientras hacemos clic en **OK**, observar el cambio en el cuadro de texto.
El nombre “Samantha” aparecerá en el cuadro de texto después de que se cierre el cuadro de mensaje, aunque no se haya escrito ningún código que asigne específicamente un nuevo valor a la propiedad **TextBox1.Text**.

Archivos de solución

Los archivos de solución para esta práctica se encuentran en la carpeta **ByRefTest\Solution** dentro del archivo **practs04.zip**.

Cómo utilizar los procedimientos Function

■ Invocar una función

- Incluir el nombre de la función y los argumentos en el lado derecho de una instrucción de asignación

```
Dim celsiusTemperature As Single
celsiusTemperature = FtoC(80)
```

- Utilizar el nombre de la función en una expresión

```
If FtoC(userValue) < 0 Then
    ...
End If
```

Introducción

Un procedimiento **Function** se diferencia de un procedimiento **Sub** en que el primero puede devolver un valor al procedimiento de llamada.

Invocar una función

Invocamos un procedimiento **Function** incluyendo su nombre y sus argumentos en el lado derecho de una instrucción de asignación o en una expresión. Piense en la siguiente función, que convierte una temperatura en Fahrenheit a una temperatura en Celsius.

```
Function FtoC(ByVal temperature As Single) As Single
    ' Convert Fahrenheit to Celsius
    FtoC = (temperature - 32.0) * (5 / 9)
End Function
```

Las siguientes llamadas de ejemplo muestran cómo podríamos invocar esta función:

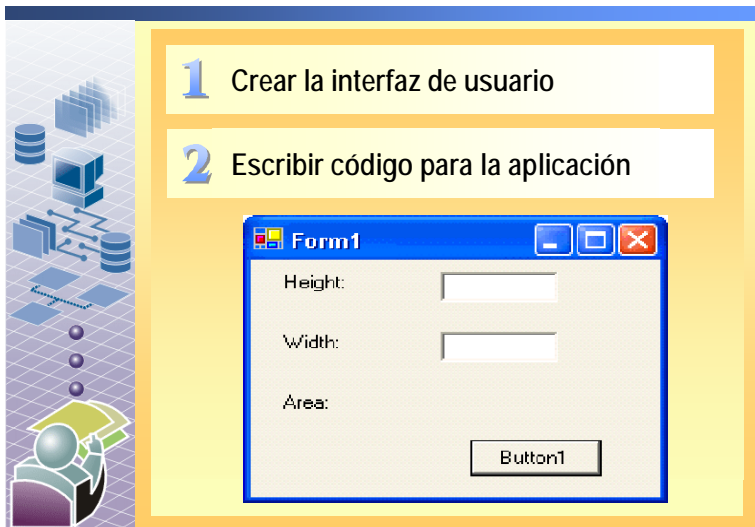
```
Dim celsiusTemperature As Single
celsiusTemperature = FtoC(80)
' Call the procedure by including its name and arguments on
' the right side of an assignment statement. In this call,
' the value 80 is passed to the FtoC function, and the
' value returned is assigned to celsiusTemperature.

If FtoC(userValue) < 0 Then . . .
' Call the procedure by using it in an expression. In this
' call, the FtoC function is used as part of an expression.
End If
```

Flujo de código

Cada vez que se invoca la función se ejecutan sus instrucciones, empezando por la primera instrucción ejecutable tras la instrucción **Function** y finalizando con la primera instrucción **End Function**, **Exit Function** o **Return** encontrada.

Práctica: utilización del valor devuelto de una función



En este ejercicio, crearemos una aplicación sencilla que calcula el área de un rectángulo, dada su altura (*height*) y anchura (*width*).

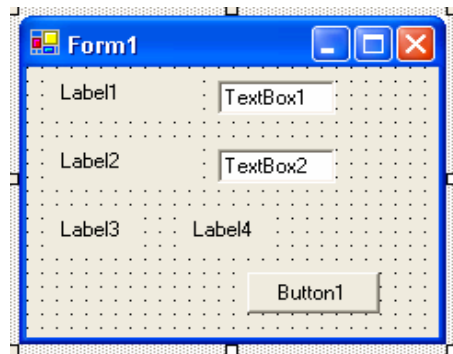
En primer lugar, crearemos el interfaz de usuario para la aplicación. El usuario escribirá valores para la altura y la anchura en dos cuadros de texto de un formulario y hará clic en un botón para calcular el área del rectángulo en función de las dimensiones introducidas. El resultado se mostrará como una etiqueta en el formulario.

A continuación, escribiremos el código para la aplicación. Invocaremos a la función **Area** que creamos en la primera práctica de este módulo (*Crear una función en un módulo*), pasaremos argumentos a la función y devolveremos un valor.

⚡ Crear el interfaz de usuario

1. Abrir el proyecto CalculateArea que creamos en la práctica *Crear una función en un módulo*. Si no finalizamos esa práctica, abrir la solución CalculateArea.sln desde FunctionReturnValue\Starter dentro del archivo practs04.zip y familiaricémonos con el formulario y con el módulo del proyecto.
2. Abrir Form1 en la vista de Diseño.

3. Abrir el Cuadro de herramientas. Añadir a nuestro formulario cuatro controles **Label**, dos controles **TextBox** y un control **Button**. Organizar los controles para que nuestro formulario tenga un aspecto similar al de la siguiente figura:



4. Establecer las propiedades para los controles como se muestra en la siguiente tabla:

| Control | Propiedad | Nuevo valor |
|----------|-------------|------------------|
| Label1 | Text | Height |
| Label2 | Text | Width |
| Label3 | Text | Area |
| Label4 | Text | <i>en blanco</i> |
| TextBox1 | Text | <i>en blanco</i> |
| TextBox2 | Text | <i>en blanco</i> |

✍ Escribir código para la aplicación

1. Añadir un controlador de eventos para el evento **Button1_Click**. En el controlador de eventos:
 - a. Invocar la función **Area**.
 - b. Pasar las propiedades **Text** de TextBox1 y TextBox2 como argumentos.
 - c. Asignar el valor devuelto a la propiedad **Text** de Label4.

Nuestro código debería ser similar al siguiente:

```
Private Sub Button1_Click(...)
    Label4.Text = Area(TextBox1.Text, TextBox2.Text)
End Sub
```

2. Ejecutar la aplicación.
3. Escribir valores numéricos en los cuadros de texto **Height** y **Width**.
4. Hacer clic en **Button1** y verificar que aparece la respuesta esperada.
5. Cerrar la aplicación.

Archivos de solución

Los archivos de solución para esta práctica están ubicados en la carpeta FunctionReturnValue\Solution dentro del archivo practs04.zip.

Cómo pasar matrices a procedimientos

■ Una matriz se pasa igual que otros argumentos:

```
Sub PassArray(ByVal testScores As Integer( ))
    ...
End Sub

Dim scores( ) As Integer = {80, 92, 73}
PassArray(scores)
```

■ Declarar una matriz parámetro:

```
Sub StudentScores(ByVal name As String, ByVal _
    ParamArray scores( ) As String)
    ' Statements for Sub procedure
End Sub
```

■ Invocar un procedimiento con una matriz parámetro:

```
StudentScores("Anne", "10", "26", "32", "15", "22", "16")
```

Introducción

Podemos pasar matrices como argumentos a un procedimiento igual que otros argumentos. Visual Basic .NET también proporciona la palabra clave **ParamArray** para declarar una matriz de parámetros en la definición de parámetros de un procedimiento.

Pasar matrices

Podemos pasar matrices unidimensionales o multidimensionales a procedimientos del mismo modo que pasamos otros argumentos.

El siguiente ejemplo muestra cómo pasar una matriz unidimensional a un procedimiento:

```
Sub PassArray(ByVal testScores As Integer( ))
    ...
End Sub

Dim scores( ) As Integer = {80, 92, 73}
PassArray(scores)
```

El siguiente ejemplo muestra cómo pasar una matriz bidimensional a un procedimiento:

```
Sub Pass2DArray(ByVal rectangle As Integer(,))
    ...
End Sub

Dim rectangle(,) As Integer = {{12, 1}, {0, 12}}
Pass2DArray(rectangle)
```

Uso de ParamArray

Normalmente, no podemos invocar un procedimiento con más argumentos de los especificados en su declaración. Cuando necesitamos un número indefinido de argumentos, podemos declarar una matriz de parámetros, que permite que un procedimiento acepte una matriz de valores para un argumento. No es necesario conocer el número de elementos de la matriz de parámetros cuando definimos el procedimiento. El tamaño de la matriz está determinado de forma individual por cada invocación al procedimiento.

Utilizamos la palabra clave **ParamArray** para denotar una matriz de parámetros. Esta palabra clave indica que el argumento de un procedimiento es una matriz opcional de elementos de un tipo especificado. Se aplican las siguientes reglas:

- Un procedimiento sólo puede tener una matriz de parámetros, y debe ser el último argumento de la definición del procedimiento.
- La matriz de parámetros debe pasarse por valor. Es una buena práctica de programación incluir explícitamente la palabra clave **ByVal** en la definición del procedimiento.
- El código dentro del procedimiento debe tratar la matriz de parámetros como una matriz unidimensional, siendo cada elemento de la misma el mismo tipo de datos que el tipo de datos **ParamArray**.
- La matriz de parámetros es automáticamente opcional. Su valor predeterminado es una matriz unidimensional vacía del tipo de elemento del parámetro de la matriz.
- Todos los argumentos que preceden a la matriz de parámetros deben ser obligatorios. La matriz de parámetros debe ser el único argumento opcional.

Invocar un procedimiento con un argumento de matriz de parámetros

Cuando invocamos un procedimiento con un argumento de matriz de parámetros, podemos pasar alguna de las opciones siguientes para la matriz de parámetros:

- Nada. Es decir, podemos omitir el argumento **ParamArray**. En este caso, se pasa al procedimiento una matriz vacía. También podemos pasar la palabra clave **Nothing**, produciendo el mismo efecto.
- Una lista de un número indefinido de argumentos, separados por comas. El tipo de datos de cada argumento debe ser implícitamente convertible al tipo de elemento **ParamArray**.
- Una matriz con el mismo tipo de elemento que la matriz de parámetros.

Ejemplo de declaración ParamArray

El siguiente código muestra cómo podemos definir un procedimiento con una matriz de parámetros:

```
Sub StudentScores(ByVal name As String, ByVal ParamArray _
    scores( ) As String)
    ' Statements for Sub procedure
End Sub
```

Ejemplos de invocaciones a un procedimiento con una matriz de parámetros

Los siguientes ejemplos muestran invocaciones posibles a **StudentScores**.

```
StudentScores("Anne", "10", "26", "32", "15", "22", "16")
```

```
StudentScores("Mary", "High", "Low", "Average", "High")
```

```
Dim JohnScores( ) As String = {"35", "Absent", "21", "30"}  
StudentScores("John", JohnScores)
```

Cómo crear un Sub Main

- Sub Main: Punto de inicio de la aplicación
- Application.Run: Inicia la aplicación
- Application.Exit: Cierra la aplicación

Introducción

Para abrir y cerrar una aplicación, la biblioteca de clases del .NET Framework proporciona la clase **Application**. La clase **Application** proporciona métodos (procedimientos) y propiedades para gestionar una aplicación, incluyendo métodos para abrir y cerrar una aplicación, métodos para procesar mensajes de Microsoft Windows®, y propiedades para obtener información sobre una aplicación.

El procedimiento Sub Main

Cuando creamos aplicaciones con la plantilla Aplicación para Windows en Visual Basic .NET, Visual Basic crea automáticamente un procedimiento **Sub** oculto denominado **Sub Main** para la clase **Form**. Este procedimiento se utiliza como punto de inicio para nuestra aplicación.

Crear un nuevo Sub Main

En el procedimiento **Sub Main**, Visual Basic .NET invoca el método **Application.Run** para iniciar la aplicación. Podemos cambiar este comportamiento creando nuestro propio **Sub Main** y convirtiéndolo en el objeto de inicio. Podemos crear **Sub Main** en un módulo o en otra clase. Después de crear un **Sub Main**, necesitamos hacer de este nuevo procedimiento el objeto de inicio utilizando la ventana Propiedades.

⚡ Cambiar el objeto de inicio a Sub Main

1. Si el Explorador de soluciones no está abierto, en el menú **Ver**, hacer clic en **Explorador de soluciones**.
2. En el Explorador de soluciones, hacer clic con el botón derecho en el nombre del proyecto y, a continuación, en **Propiedades**.
3. En el panel izquierdo, debajo de **Propiedades comunes**, verificar que está seleccionado **General**.
4. En la lista **Objeto inicial**, hacer clic en **Sub Main** para convertir este procedimiento el nuevo objeto de inicio de nuestro proyecto.

**Utilización de
Application.Exit**

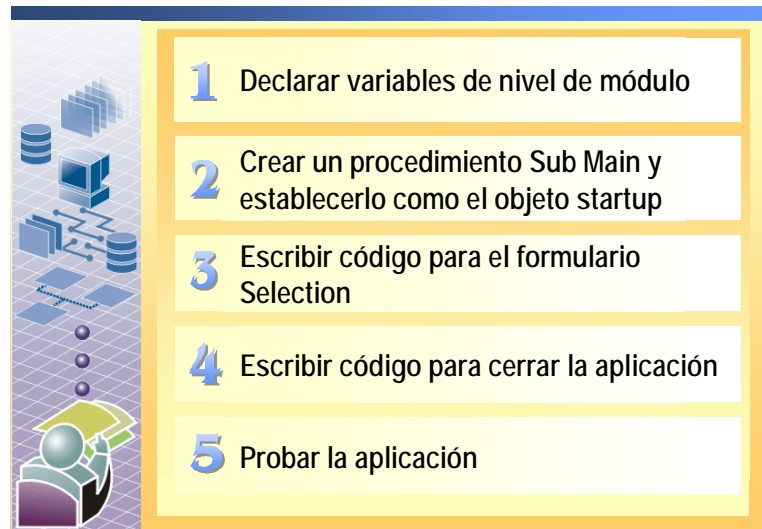
Para cerrar una aplicación, invocamos el método **Application.Exit** utilizando la siguiente sintaxis:

```
Application.Exit( )
```

Por ejemplo, podemos insertar este código en el controlador de eventos **Click** de un botón. Cuando el usuario haga clic en el botón, la aplicación se cerrará.

Nota Los eventos **Form.Closed** y **Form.Closing** no se lanzan cuando se invoca el método **Application.Exit** para cerrar la aplicación. Si hay código en estos eventos que deba ser ejecutado, invocar el método **Form.Close** para cada formulario abierto individualmente antes de invocar el método **Application.Exit**.

Práctica: Crear un Sub Main



En esta práctica, generaremos código para una aplicación formada por tres formularios, y crearemos un **Sub Main** como objeto de inicio para el proyecto.

⚡ Abrir el proyecto

- Abrir el proyecto MultipleForms.sln, ubicado en la carpeta Forms\Starter dentro del archivo practs04.zip y familiaricémonos con los formularios y los módulos del proyecto.

⚡ Declarar variables

- Declarar las siguientes variables en el archivo de módulo *Starter*.

| Nombre de la variable | Modificador de acceso | Tipo de datos |
|-----------------------|-----------------------|------------------|
| <i>carLoanForm</i> | Friend | CarLoan |
| <i>homeLoanForm</i> | Friend | HomeLoan |
| <i>selectionForm</i> | Private | Selection |

Nuestro código debería ser similar al siguiente:

```
Friend carLoanForm As CarLoan
Friend homeLoanForm As HomeLoan
Private selectionForm As Selection
```


✍ Crear un procedimiento Sub Main

1. Declarar un procedimiento **Sub Main** público en el módulo Starter. Nuestro código debería ser similar al siguiente:

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

2. Completar el cuerpo del procedimiento **Sub Main**.
 - a. Crear una nueva instancia del formulario Selection y asignarlo a *selectionForm*.
 - b. Invocar el método **selectionForm.Show** para visualizar el formulario.
 - c. Iniciar la aplicación invocando el método **Application.Run**.

Nuestro código debería ser similar al siguiente:

```
Public Sub Main( )
    selectionForm = New Selection( )
    selectionForm.Show( )
    Application.Run( )
End Sub
```

3. Establecer **Sub Main** como objeto de inicio de la aplicación.

✍ Escribir código para el formulario Selection

1. Abrir el Editor de código para el formulario Selection.
2. Añadir un controlador de eventos **Click** para el botón **Next**.
3. Escribir código para el controlador de eventos.
 - a. Si está seleccionado **CarLoanRadioButton**, crear una nueva instancia del formulario CarLoan. Nuestro código debería ser similar al siguiente:

```
If CarLoanRadioButton.Checked Then
    carLoanForm = New CarLoan( )
```

Nota Estudiaremos con mayor profundidad las instrucciones **If...Then** más adelante.

- b. Utilizar el método **Show** para visualizar el formulario CarLoan.
- c. Utilizar el método **Close** para cerrar el formulario Selection, como sigue:

```
carLoanForm.Show( )
Me.Close( )
```

- d. Si está seleccionado **HomeLoanRadioButton**, crear una nueva instancia del formulario HomeLoan. Nuestro código debería ser similar al siguiente:

```
ElseIf HomeLoanRadioButton.Checked Then
    homeLoanForm = New HomeLoan( )
```

- e. Utilizar el método **Show** para visualizar el formulario HomeLoan.

- f. Utilizar el método **Close** para cerrar el formulario Selection, como sigue:

```
homeLoanForm.Show( )
Me.Close( )
```

- g. Si no se selecciona ninguna opción, utilizar un cuadro de mensaje para indicar al usuario que realice una selección. Nuestro código debería ser similar al siguiente:

```
Else
    MessageBox.Show("Please select a loan type", _
        "Loan Type", MessageBoxButtons.OK, _
        MessageBoxIcon.Error)
End If
```

✍ Escribir código para cerrar la aplicación

1. Crear un controlador de eventos **Click** para el botón **Exit** del formulario Selection. Invocar **Application.Exit** para cerrar la aplicación.
2. Crear un controlador de eventos **Click** para el botón **Done** del formulario HomeLoan. Invocar **Application.Exit** para cerrar la aplicación.
3. Crear un controlador de eventos **Closing** para el formulario HomeLoan. Invocar **Application.Exit** para cerrar la aplicación.
4. Crear un controlador de eventos **Click** para el botón **Done** del formulario CarLoan. Invocar **Application.Exit** para cerrar la aplicación.
5. Crear un controlador de eventos **Closing** para el formulario CarLoan. Invocar **Application.Exit** para cerrar la aplicación.

Nota Es importante poder invocar **Application.Exit** en cualquier punto de la aplicación para que el usuario pueda cerrar la aplicación. Si no se invoca el código de cierre apropiado, la aplicación seguirá ejecutándose, aunque el formulario cerrado ya no esté accesible. En una aplicación más compleja, el código de cierre podría incluir código para guardar los cambios, para cerrar conexiones a datos, para confirmar que el usuario desea cerrar la aplicación, etc.

✍ Probar la aplicación

1. Ejecutar la aplicación. Hacer clic en **Car Loan** y en el botón **Next**. Comprobar que se abre el formulario Car Loan.
2. Salir de la aplicación haciendo clic en el botón **Close** de la esquina superior derecha del formulario.
3. Ejecutar de nuevo la aplicación. Hacer clic en **Home Loan** y en el botón **Next**. Comprobar que se abre el formulario Home Loan.
4. Salir de la aplicación haciendo clic en el botón **Done**.
5. Ejecutar de nuevo la aplicación. Hacer clic en el botón **Next** sin pulsar en un tipo de crédito (loan). Comprobar si aparece un cuadro de mensaje indicando al usuario que seleccione un tipo de crédito (*loan*).
6. Salir de la aplicación utilizando el botón **Exit** del formulario Selection.

Archivos de solución

Los archivos de solución para esta práctica están ubicados en Forms\Solution dentro del archivo practs04.zip.

Lección: uso de funciones predefinidas

- Cómo utilizar la función **InputBox**
- Cómo utilizar las funciones de fecha y hora
- Cómo utilizar las funciones **String**
- Cómo utilizar las funciones **Format**
- Cómo utilizar las funciones **Financial**

Introducción

La biblioteca del entorno de ejecución de Visual Basic proporciona numerosas funciones predefinidas que podemos utilizar en nuestro código. Estas funciones se invocan del mismo modo en que invocamos a nuestras propias funciones.

En este módulo, estudiaremos la función **InputBox**, las funciones de fecha y hora, las funciones de cadena, las funciones de formato y las funciones financieras. En “Miembros de la biblioteca del entorno de ejecución de Visual Basic”, en la documentación de Visual Studio .NET, encontraremos una lista completa de las funciones predefinidas.

Estructura de la lección

Esta lección incluye los siguientes temas y actividades:

- Cómo utilizar la función **InputBox**
- Cómo utilizar las funciones **Date** y **Time**
- Cómo utilizar las funciones **String**
- Cómo utilizar las funciones **Format**
- Cómo utilizar las funciones **Financial**
- Práctica: Examen de funciones predefinidas

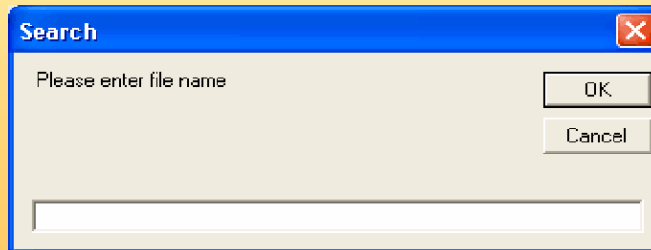
Objetivo de la lección

En esta lección, aprenderemos a utilizar funciones predefinidas en el código de nuestra aplicación, incluyendo la función **InputBox**, funciones de fecha y hora, funciones de cadena, funciones de formato y funciones financieras.

Cómo utilizar la función InputBox

- Muestra un mensaje en un cuadro de diálogo y devuelve al usuario input en una cadena

```
Dim FileName As String
FileName = InputBox("Please enter file name","Search")
```



Introducción

La función **InputBox** es una función predefinida que proporciona una forma sencilla de interactuar con los usuarios. La función **InputBox** muestra un cuadro de diálogo con un mensaje, espera a que el usuario introduzca texto o haga clic en un botón y devuelve una cadena con el contenido del cuadro de texto.

Parámetros

La siguiente declaración de función muestra los parámetros de la función **InputBox**:

```
Public Function InputBox(ByVal Prompt As String, _
    Optional ByVal Title As String = "", _
    Optional ByVal DefaultResponse As String = "", _
    Optional ByVal XPos As Integer = -1, _
    Optional ByVal YPos As Integer = -1 ) As String
```

Como podemos ver, los argumentos **Title**, **DefaultResponse**, **XPos** e **YPos** son opcionales. La siguiente tabla explica los valores predeterminados que se aplicarían en caso de escoger no pasar uno de estos argumentos opcionales a la función **InputBox**.

| Parámetro | Definición | Valor predeterminado |
|------------------------|--|---|
| Title | Texto que aparece en la barra de título | El nombre de la aplicación |
| DefaultResponse | El valor que se muestra en el cuadro de texto como valor predeterminado si el usuario no proporciona una entrada | El cuadro de texto se mostrará vacío |
| XPos | Especifica la distancia entre el borde izquierdo del cuadro de diálogo y el extremo izquierdo de la pantalla | El cuadro de diálogo se centrará horizontalmente |
| YPos | Especifica la distancia entre el borde superior del cuadro de diálogo y la parte superior de la pantalla | El cuadro de diálogo se posicionará verticalmente aproximadamente a un tercio del alto del total de la pantalla |

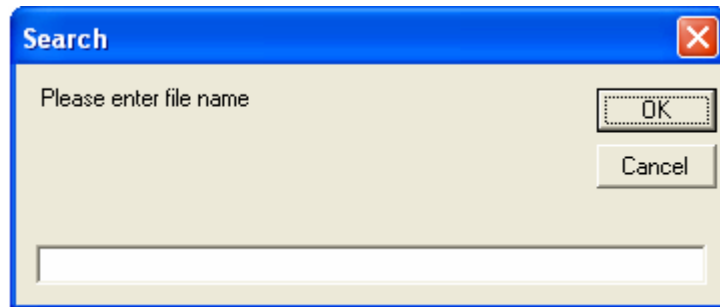
Ejemplo de utilización de la función InputBox

El siguiente código crea un cuadro de entrada con el título **Search**, que solicita al usuario que introduzca un nombre de archivo y almacena la respuesta del usuario en una variable denominada *FileName*.

```
Dim FileName As String  
FileName = InputBox("Please enter file name", "Search")
```

Resultado

La siguiente ilustración de pantalla muestra el cuadro de entrada creada por el código anterior.



Nota Encontrará más información sobre la función **InputBox** en “Función **InputBox**”, en la documentación de Visual Studio .NET.

Cómo utilizar las funciones de fecha y hora

- Realizan cálculos y operaciones que implican fechas y horas

- Ejemplos:

- **DateAdd**: Añade o sustrae un intervalo de tiempo específico a una fecha

```
DateAdd(DateInterval.Day, 10, billDate)
```

- **DateDiff**: Determina cuántos intervalos de tiempo especificados existen entre dos valores de fecha/hora

```
DateDiff(DateInterval.Day, Now, secondDate)
```

Introducción

Visual Basic proporciona numerosas funciones de fecha y hora que podemos utilizar en nuestras aplicaciones. En esta sección, estudiaremos cómo utilizar dos funciones predefinidas para realizar cálculos y operaciones que implican fechas y horas.

Uso de la función DateAdd

Podemos utilizar la función **DateAdd** para añadir o sustraer un intervalo de tiempo específico a una fecha. Pasamos a la función la fecha y la información sobre el intervalo, y la función **DateAdd** devuelve un valor **Date** que contiene el valor de fecha y hora, al que se ha añadido o sustraído un intervalo de tiempo especificado.

Parámetros

La función **DateAdd** tiene tres parámetros, ninguno de ellos opcional, que se muestran en la siguiente tabla:

| Parámetro | Definición |
|------------------|---|
| Interval | Valor de enumeración DateInterval o expresión String que representa el intervalo de tiempo que se desea añadir El apartado “La función DateAdd ” de la documentación de Visual Studio .NET contiene información sobre posibles configuraciones del argumento Interval . |
| Number | Expresión en punto flotante que representa el número de intervalos que se desea agregar. Number puede ser positivo (para obtener valores de fechas u horas futuras) o negativo (para obtener valores de fechas u horas pasadas). |
| DateValue | Expresión que representa la fecha y la hora a la que debe agregarse el intervalo. |

Ejemplo de utilización de la función DateAdd

El siguiente código utiliza la función **DateAdd** para calcular la fecha de vencimiento de una factura a partir de la fecha de facturación. En este escenario, la fecha de vencimiento de la factura es 20 días después de la fecha de factura. Por tanto, **Interval** es **DateInterval.Day** y **Number** es 20.

```
fechaFactura = #12/31/2000#
DateAdd(DateInterval.Day, 20, fechaFactura)
```

Utilización de la función DateDiff

Podemos utilizar la función **DateDiff** para determinar cuántos intervalos de tiempo especificados existen entre dos valores de fecha/hora. Por ejemplo, podría utilizarse **DateDiff** para calcular el número de días entre dos fechas o el número de semanas entre hoy y final del año.

Parámetros

La función **DateDiff** tiene cinco parámetros, dos de los cuales son opcionales. Estos parámetros se muestran en la tabla siguiente:

| Parámetro | Definición |
|---------------------------------|---|
| Interval | <p>Valor de enumeración DateInterval o expresión String que representa el intervalo de tiempo que se desea utilizar como unidad de diferencia entre Date1 y Date2.</p> <p>El apartado “Función DateDiff” en la documentación de Visual Studio .NET contiene información sobre posibles configuraciones del argumento Interval.</p> |
| Date1, Date2 | <p>Los dos valores de fecha/hora que se desea utilizar en el cálculo. El valor de Date1 se sustrae del valor de Date2 para obtener la diferencia. Ninguno de los valores se cambia en el programa de llamada.</p> |
| DayOfWeek (Opcional) | <p>Valor escogido de la enumeración FirstDayOfWeek que especifica el primer día de la semana.</p> <p>FirstDayOfWeek.Sunday es el valor predeterminado.</p> |
| WeekOfYear (Opcional) | <p>Valor escogido de la enumeración FirstWeekOfYear que especifica la primera semana del año.</p> <p>FirstWeekOfYear.Jan1 es el valor predeterminado.</p> |

Ejemplo de utilización de la función DateDiff

Este ejemplo utiliza la función **DateDiff** para mostrar el número de días entre una fecha determinada y la fecha actual.

```
Dim firstDate, msg As String
Dim secondDate As Date
' Declare variables
firstDate = InputBox("Enter a date")
' Get a given date from the user
secondDate = CDate(firstDate)
msg = "Days from today: " & DateDiff(DateInterval.Day, Now, _
    secondDate)
MessageBox.Show(msg)
' Create a message box which uses the DateDiff function and
' displays the number of days between a given date and the
' current date
```

Nota Encontrará información sobre otras funciones de fecha y hora en la sección **Microsoft.VisualBasic.DateandTime** de “*Miembros de la biblioteca run-time de Visual Basic*” en la documentación de Visual Studio .NET.

Cómo utilizar las funciones String

- Extraen sólo una parte determinada de una cadena
- Devuelven información sobre una cadena
- Muestran información de un formato determinado
- Ejemplos:

- Trim

```
NewString = Trim(MyString)
```

- Len

```
Length = Len(customerName)
```

- Left

```
Microsoft.VisualBasic.Left(customerName, 5)
```

Introducción

En muchos casos, las cadenas (*strings*) requieren algún tipo de manipulación, formateo o evaluación. Por ejemplo, el nombre de una persona puede escribirse con el apellido delante del nombre de pila, o un archivo puede contener campos separados por comas. Las funciones String de Visual Basic pueden analizar y manipular cadenas en las aplicaciones. Estas funciones se utilizan para devolver información sobre una cadena, extraer únicamente una parte de la cadena, o mostrar información en un determinado formato.

Uso de la función Trim

Podemos utilizar la función **Trim** para eliminar los espacios iniciales y finales de una cadena específica.

Ejemplo de utilización de Trim

El siguiente ejemplo muestra cómo utilizar la función **Trim** para devolver una cadena que contenga una copia de una cadena específica sin espacios iniciales ni finales:

```
Dim MyString, NewString As String
' Initialize string
MyString = " 1234 Street "
' NewString = "1234 Street"
NewString = Trim(MyString)
```

Uso de la función Len

La función **Len** puede utilizarse para encontrar el número de caracteres de una cadena o el número de bytes necesarios para almacenar una variable.

El siguiente código muestra la declaración para la función **Len**. El parámetro *Expression* de esta declaración es cualquier expresión de cadena o nombre de variable válidos.

```
Public Shared Function Len(ByVal Expression As datatype) As _
    Integer
```


Ejemplo de utilización de Len

En el siguiente código, la función **Len** devuelve el número de caracteres de una cadena:

```
Dim customerName As String
Dim length As Integer
customerName = InputBox("What is your name?")
length = Len(customerName)
```

Uso de la función Left

Podemos utilizar la función **Left** para devolver un número especificado de caracteres desde el lado izquierdo de una cadena.

El siguiente código muestra la declaración de función para la función **Left**. El parámetro *Str* de esta declaración es la expresión de cadena de la que la función devolverá los caracteres más a la izquierda. El parámetro *Length* es un entero que indica cuántos caracteres devolver.

```
Public Shared Function Left(ByVal Str As String, _
    ByVal Length As Integer) As String
```

- Si el valor de *Length* es 0, se devuelve una cadena de longitud cero (" ").
- Si el valor de *Length* es mayor o igual al número de caracteres de *Str*, se devuelve toda la cadena.
- Si utilizamos esta función en un formulario Windows Form o en cualquier clase que tenga una propiedad **Left**, debemos invocar a la función utilizando su nombre completamente cualificado: **Microsoft.VisualBasic.Left**.

Ejemplo de utilización de Left

En el siguiente código, la función **Left** devuelve cinco caracteres del lado izquierdo de una cadena:

```
Microsoft.VisualBasic.Left(customerName, 5)
```

Ejemplo de utilización de Len y Left

En el siguiente código, las funciones **Len** y **Left** se utilizan juntos para eliminar la extensión del nombre del archivo (los cuatro últimos caracteres) de un nombre de archivo:

```
fileName = Left(fileName, Len(fileName) - 4)
```

Nota La sección **Microsoft.VisualBasic.Strings** de “*Miembros de la biblioteca run-time de Visual Basic*” en la documentación de Visual Studio .NET contiene más información sobre otras funciones String.

Cómo utilizar funciones Format

- Formatean números, fechas y horas según estándares aceptados
- Muestran formatos regionales sin re-coding para nacionalidades o regiones
- Ejemplos:
 - FormatCurrency

```
FormatCurrency(amountOwed, , , TriState.True, TriState.True)
```

- FormatDateTime

```
FormatDateTime(myDate, DateFormat.LongDate)
```

Introducción

Existen varios formatos aceptados universalmente para números, fechas y horas. Visual Basic ofrece una gran flexibilidad en visualizar formatos de números además de formatos de fecha y hora. Un beneficio añadido es que los formatos regionales para números, fechas y horas se presentan fácilmente sin codificar nuevamente para cada nacionalidad o región.

En esta sección, estudiaremos cómo utilizar dos funciones predefinidas para formatear.

Uso de la función FormatCurrency

La función **FormatCurrency** puede utilizarse para devolver una expresión con formato de moneda que utiliza el símbolo de moneda definido en el panel de control del sistema. Por ejemplo, se puede utilizar **FormatCurrency** para formatear el importe debido en una factura.

Parámetros

La función **FormatCurrency** tiene cinco parámetros, cuatro de los cuales son opcionales. Cuando se omite uno o más argumentos opcionales, se utilizan valores que se ajustan a la configuración regional predeterminada del equipo. La siguiente tabla describe los parámetros de la función **FormatCurrency**:

| Parámetro | Definición |
|--|---|
| Expression | Expresión que debe formatearse. |
| NumDigitsAfterDecimal (Opcional) | Valor numérico que indica cuántos lugares se muestran a la derecha del decimal. El valor predeterminado es -1, que indica que se utiliza la configuración regional del equipo. |
| IncludeLeadingDigit (Opcional) | Indica si se muestra un cero al principio para valores fraccionados. La configuración utiliza la enumeración Tristate para configurar el parámetro como <i>true</i> , <i>false</i> , o <i>use default</i> , por ejemplo, TriState.True , TriState.False o TriState.UseDefault . |

(continuación)

| Parámetro | Definición |
|--|---|
| UseParensForNegativeNumbers (Opcional) | Indica si los valores negativos han de mostrarse entre paréntesis o no. La configuración utiliza la enumeración <i>TriState</i> para configurar el parámetro como <i>true</i> , <i>false</i> o <i>use default</i> , por ejemplo, TriState.True , TriState.False o TriState.UseDefault . |
| GroupDigits (Opcional) | Indica si los números se agrupan utilizando el delimitador de grupos especificado en la configuración regional del equipo; por ejemplo, utilizar una coma cada tres dígitos a la izquierda de una coma decimal. La configuración utiliza la enumeración <i>TriState</i> para configurar el parámetro como <i>true</i> , <i>false</i> o <i>use default</i> , por ejemplo, TriState.True , TriState.False o TriState.UseDefault . |

Ejemplo de utilización
de la función
FormatCurrency

El siguiente ejemplo utiliza la función **FormatCurrency** para formatear una cadena que representa **importeDebido**:

```
Dim importeDebido As Double = 4456.43
Dim myString As String
myString = FormatCurrency(importeDebido, , , TriState.True, _
    TriState.True)

' Returns "$4,456.43" when regional settings are set to
' English (United States)
```

Uso de la función
FormatDateTime

La función **FormatDateTime** puede utilizarse para formatear una expresión como fecha u hora. Por ejemplo, podemos utilizar **FormatDateTime** para cambiar la expresión **5/21/01** por la expresión **Monday, May 21, 2001**.

Parámetros

La función **FormatDateTime** tiene dos parámetros, como se muestra en la siguiente tabla:

| Parámetro | Definición |
|----------------------------------|---|
| Expression | Expresión que debe formatearse. |
| NamedFormat (Opcional) | Indica qué formato utilizar. Si se omite, se utilizará GeneralDate . |

La siguiente tabla muestra una posible configuración del parámetro **NamedFormat**:

| Constante | Descripción |
|-------------------------------|--|
| DateFormat.GeneralDate | Muestra una fecha y/u hora. Si hay una fecha, la presenta como fecha corta. Si hay una hora, la presenta en formato largo. Si las dos partes están presentes, se muestran ambas. |
| DateFormat.LongDate | Muestra una fecha en el formato largo especificado en la configuración regional del equipo. |
| DateFormat.ShortDate | Muestra una fecha en el formato corto especificado en la configuración regional del equipo. |
| DateFormat.LongTime | Muestra una hora en el formato de hora especificado en la configuración regional del equipo. |
| DateFormat.ShortTime | Muestra una hora en el formato de 24 horas (hh:mm). |

Ejemplo de utilización de la función **FormatDateTime**

El siguiente ejemplo utiliza la función **FormatDateTime** para formatear la expresión **5/21/01**.

```
Dim myDate As DateTime = #5/21/01#
Dim myString As String
myString = FormatDateTime(myDate, DateFormat.LongDate)
' Returns "Monday, May 21, 2001" when regional settings are
' set to English (United States)
```

Nota La sección **Microsoft.VisualBasic.Strings** de “*Miembros de la biblioteca run-time de Visual Basic*” en la documentación de Visual Studio .NET contiene más información sobre otras funciones para dar formato.

Cómo utilizar las funciones Financial

- Realizan cálculos y operaciones que implican finanzas; por ejemplo, tipos de interés

- Ejemplos:

- Pmt

```
payment = Pmt(0.0083, 24, -5000, 0, DueDate.BegOfPeriod)
```

- Rate

```
ratePerPeriod = Rate(24, 228, -5000, 0, DueDate.BegOfPeriod, _  
0.8)*100
```

Introducción

Visual Basic ofrece varias funciones financieras que podemos utilizar en nuestras aplicaciones. En esta sección, estudiaremos cómo utilizar dos funciones predefinidas para realizar cálculos y operaciones que implican dinero.

Uso de la función Pmt

Podemos utilizar la función **Pmt** para calcular el pago de una anualidad basado en pagos fijos periódicos y un tipo de interés fijo. Una *anualidad* es una serie de pagos fijos que se realizan durante un periodo de tiempo, como una hipoteca sobre una casa o el pago de un crédito para la compra de un coche.

Parámetros

La función **Pmt** tiene cinco parámetros, dos de los cuales son opcionales. Para todos los argumentos, el efectivo abonado (como depósitos de ahorro) se representa con números negativos. El efectivo recibido (como cheques de dividendos) se representa con números positivos. La siguiente tabla describe los cinco parámetros de la función **Pmt**.

| Parámetro | Definición |
|-------------|--|
| Rate | Tipo de interés por periodo, expresado como Double . Por ejemplo, si obtiene un crédito para comprar un automóvil a una tasa porcentual anual (TAE) del 10 por ciento, y realiza pagos mensuales, el tipo por periodo será 0,1/12 ó 0,0083. |
| NPer | Número total de periodos de pago de la anualidad, expresado como Double . Por ejemplo, si realiza pagos mensuales correspondientes a un crédito para adquirir un automóvil a dos años, el crédito tendrá un total de 2 * 12 (ó 24) periodos de pago. NPer debe calcularse utilizando periodos de pago expresados en las mismas unidades que los periodos utilizados en Rate . Por ejemplo, si Rate es por mes, NPer debe ser por mes. |
| PV | Valor actual de una serie de pagos que se realizarán en el futuro, expresado como Double . Por ejemplo, cuando compramos un coche, la cantidad del crédito es el valor actual de los pagos mensuales que realizaremos. |

(continuación)

| Parámetro | Definición |
|--------------------------|--|
| FV (Opcional) | Valor futuro, o saldo, que se desea tener una vez realizado el pago final, expresado como Double . El valor predeterminado, si no se expresa ningún otro, es 0. |
| Due (Opcional) | Indica cuándo vencen los pagos. Puede ser al final del periodo (especificado como DueDate.EndOfPeriod) o al principio del periodo (especificado como DueDate.BegOfPeriod). El valor predeterminado, si no se expresa ningún otro, es DueDate.EndOfPeriod . |

Ejemplo de utilización de la función **Pmt**

El siguiente código utiliza la función **Pmt** para calcular el pago mensual de un crédito de 24 meses de 5000 dólares al 10% TAE.

```
payment = Pmt(0.0083, 24, -5000, 0, DueDate.BegOfPeriod)
```

Uso de la función **Rate**

La función **Rate** puede utilizarse para calcular el tipo de interés por periodo de una anualidad.

Parámetros

La función **Rate** tiene los mismos parámetros que la función **Pmt**, con las siguientes excepciones:

- Toma un argumento **Pmt** en lugar de un argumento **Rate**. **Pmt** es un argumento obligatorio que representa el pago que debe realizarse cada periodo y se expresa como **Double**.
- Toma un argumento **Guess** opcional. Es el valor estimado que será devuelto por **Rate**, expresado como **Double**. El valor predeterminado, si no se expresa ningún otro, es 0.1 (10 por ciento).


Ejemplo de utilización de la función **Rate**

El siguiente código utiliza la función **Rate** para calcular el tipo de interés por periodo de un crédito de 24 meses de 5000 dólares con pagos mensuales de 228 dólares.

```
ratePerPeriod = Rate(24, 228, -5000, 0, DueDate.BegOfPeriod, _  
0.8) * 100
```

Nota La sección **Microsoft.VisualBasic.Financial** de “*Miembros de la biblioteca run-time de Visual Basic*” en la documentación de Visual Studio .NET contiene información sobre otras funciones financieras.

Práctica: examen de las funciones predefinidas



- 1 Abrir el documento "Miembros de la biblioteca run-time de Visual Basic"
- 2 Examinar las funciones predefinidas, métodos y propiedades que pueden utilizarse en el código
- 3 Responder a cuestiones sobre funciones específicas, como InStr, Mid, y Right

En esta práctica, utilizaremos la documentación de Visual Studio para estudiar las funciones predefinidas y cómo utilizarlas en nuestro código.

⚡ Abrir Referencia del lenguaje

1. Hacer clic en **Inicio**, seleccionar **Todos los programas, Microsoft Visual Studio .NET 2003** y hacer clic en **Documentación de Microsoft Visual Studio .NET**.
2. En el menú **Ayuda**, hacer clic en **Buscar**.
3. En la ventana de búsqueda, en el cuadro **Buscar**, escribir **run-time library members**. Verificar que la búsqueda está filtrada por **Visual Basic y relacionados**, seleccionar **Buscar en títulos sólo** y hacer clic en **Buscar**.
4. En la ventana de resultados de la búsqueda, hacer doble clic en **Visual Basic Run-time Library Members**.
5. Sin hacer clic en ninguno de los enlaces de la página, examine el contenido de esta biblioteca. Observe que esta página describe funciones, métodos y propiedades que podemos utilizar en nuestro código.

🔍 Uso del contenido de la biblioteca

Utilice el documento “*Visual Basic Run-time Library Members*” y los documentos vinculados para responder a las cuestiones de la lista siguiente. Para regresar al documento “*Visual Basic Run-time Library Members*” desde otros documentos, puede utilizar el botón **Atrás** de la barra de herramientas.



1. ¿Cómo está organizado el contenido del documento “*Visual Basic Run-time Library Members*”?

El contenido está agrupado por categorías de funcionalidad.

2. ¿Para qué se utiliza la función **InStr**?

La función InStr devuelve un entero que especifica la posición inicial de la primera aparición de una cadena dentro de otra.

3. ¿Qué valor devolverá la función **InStr** si no puede localizar la segunda cadena?

La función devolverá 0.

4. ¿Cuáles son los parámetros de la función **Mid**? Indique si los parámetros son obligatorios u opcionales.

La función Mid toma dos parámetros obligatorios, Str y Start, y un parámetro opcional, Length.

5. Basándose en su respuesta a la pregunta anterior, ¿qué valor cree que asignará el siguiente código a las variables *myAnswer* y *secondAnswer*?

```
Dim myString, myAnswer, secondAnswer As String
myString = "Using Mid Function"
myAnswer = Mid(myString, 7, 3)
secondAnswer = Mid(myString, 7)
```

El valor devuelto para myAnswer es “Mid”.

El valor devuelto para secondAnswer es “Mid Function”.

-
6. En la línea siguiente de este código, añada código que utilice la función **Right** para devolver la cadena “Doe” de la cadena “John Doe”.

```
Dim myString As String = "John Doe"  
Dim subString As String
```

```
subString = Microsoft.VisualBasic.Right(myString, 3)
```

7. ¿Es **Now** una función predefinida?

No. Now es una propiedad pública de la estructura DateTime del espacio de nombres System.

8. El apartado que describe la función **Now** de *Referencia del lenguaje* incluye enlaces a otros temas de referencia. Enumere cuatro.

La sección “Vea también” de la parte inferior del tema Now (Propiedad) contiene enlaces a 13 temas. Estos temas incluyen Day (Función), Hour (Función), Month (Función) y Today (Propiedad).
