Unidad

2

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 3

Control del Programa y Vectores

Control del Programa y Vectores

En este Capítulo

- Sentencias if – else ó If – Else
- Sentencia switch de C#
- El Select Case de VB
- La sentencia for ó For
- Sentencia while de C# y las sentencias While, Do While de VB
- Sentencia do-while de C# y Do-Loop While de VB
- Casos Especiales de Control del Flujo
- Vectores
- Matrices o Vectores Bidimensionales
- Límite de un Vector
- Redimensionamiento de un Vector
- Copia de Vectores

Sentencias `if – else` ó `If – Else`

Proporciona a los programas la posibilidad de ejecutar selectivamente otras sentencias basándose en algún criterio booleano. Generalmente, la forma sencilla es

C#

```
if (expresión booleana) sentencia;
```

VB

```
If (expresión booleana) Then sentencia
```

Ejemplo

C#

```
if (a == b) Console.WriteLine("a is igual a b");
```

VB

```
If (a = b) Then Console.WriteLine("a is igual a b")
```

Este formato es válido cuando se quiere ejecutar una sola sentencia dado que la condición es verdadera. Sin embargo, si se quiere usar más de una sentencia, se debe utilizar un bloque asociado al `if` ó `If` (C# ó VB) para agruparlas. Su formato es

C#

```
if (expresión booleana)
{
      [sentencias;]
}
```

VB

```
If (expresión booleana) Then
      [sentencias]
End If
```

Ejemplo

C#

```
if (a == b)
{
    Console.WriteLine("a is igual a b");
}
```

VB

```
If (a = b) Then
      Console.WriteLine("a is igual a b")
End If
```

Cuando se quiere analizar también el caso en el cual la expresión booleana es falsa se puede utilizar la sentencia `else` ó `Else` (C# ó VB).

C#

```
if (expresión booleana) sentencia;

else sentencia;
```

VB

```
If (expresión booleana) Then
      [sentencia]
Else
      [sentencia]
End If
```

Ejemplo

C#

```
if (a > b) Console.WriteLine("a es mayor que b");
else Console.WriteLine("b es mayor que a");
```

VB

```
If (a > b) Then
      Console.WriteLine("a es mayor que b")
Else
      Console.WriteLine("b es mayor que a")
End If
```

También se pueden agrupar sentencias en la parte verdadera del condicional

C#

```
if (expresión booleana)
{
      [sentencias;]
}
else sentencia;
```

VB

```
If (expresión booleana) Then
      [sentencias]
Else
      [sentencia]
End If
```

Ejemplo

CS

```
if (a == c)
{
      Console.WriteLine("Isósceles");
      return;
```

```
}
else
{
      Console.WriteLine("Escaleno");
}
```

VB

```
  If (a = c) Then
      Console.WriteLine("Isósceles")
      Return
  Else
      Console.WriteLine("Escaleno")
  End If
```

Nuevamente, esto es útil si se ejecutan varias sentencias si la condición es verdadera y tan solo una en caso de ser falsa.

Para el caso de ejecutar varias sentencias cuando la condición es falsa y, por ejemplo, una sentencia sola si es verdadera, se puede volver a utilizar la técnica de asociar un bloque de sentencias, por ejemplo

C#

```
if (expresión booleana) sentencia;
else
{
      [sentencias;]
}
```

VB

```
If (expresión booleana) Then
      [sentencia]
Else
      [sentencias]
End If
```

El último caso a analizar es cuando se ejecutan muchas sentencias en ambas situaciones, el formato es

C#

```
if (expresión booleana)
{
      [sentencias;]
}
else
{
      [sentencias;]
}
```

VB

```
If (expresión booleana) Then
```

```
        [sentencias]
Else
        [sentencias]
End If
```

Nota: Visual Basic admite un caso especial en el cual se pueden poner varias instrucciones en una misma línea separadas por dos puntos:

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

Ejemplo

C#

```
if (b == c)
{
        Console.WriteLine("Equilátero");
        return;
}
else
{
        Console.WriteLine("Isósceles");
        return;
}
```

VB

```
If (b = c) Then
        Console.WriteLine("Equilátero")
        Return
Else
        Console.WriteLine("Isósceles")
        Return
End If
```

Existe otra forma de la sentencia `else` ó `Else` (C# ó VB), cuando se combina con un `if` ó `If` (C# ó VB) y conforman un `else if` que ejecuta una sentencia basada en otra expresión. No es más que una forma particular de combinar un `if` con un `else` (no es una sentencia diferente para C# pero si para VB) y, aunque no es recomendable porque puede inducir a errores, muchos programadores la utilizan.

Ejemplo

C#

```
namespace ifElse
{
    class Program
    {
        static void Main(string[] args)
        {
            int puntuacion = 65;
            String nota = null;
            if (puntuacion >= 90)
            {
                nota = "Sobresaliente";
```

```
            }
            else if (puntuacion >= 80)
            {
                nota = "Notable";
            }
            else if (puntuacion >= 70)
            {
                nota = "Bien";
            }
            else if (puntuacion >= 60)
            {
                nota = "Suficiente";
            }
            else
            {
                nota = "Insuficiente";
            }
            Console.WriteLine("la nota es: " + nota);

        }
    }
}
```

VB

```
    Public Sub Else_If()
        Dim puntuacion As Integer = 65
        Dim nota As String
        If (puntuacion >= 90) Then
            nota = "Sobresaliente"
        ElseIf (puntuacion >= 80) Then
            nota = "Notable"
        ElseIf (puntuacion >= 70) Then
            nota = "Bien"
        ElseIf (puntuacion >= 60) Then
            nota = "Suficiente"
        Else
            nota = "Insuficiente"
            Console.WriteLine("la nota es: " + nota.ToString())
        End If
    End Sub
```


Otro caso a tener en cuenta es el del anidamiento. Cuando se anidan sentencia `if` – `else` (C#) ó `If`–`Else` (VB), es importante tener en cuenta qué `if` ó `If` (C# ó VB)se asocia con cuál `else` ó `Else` (C# ó VB).

Para esto, se debe seguir la siguiente regla:

> *El* `else` ó `Else` (C# ó VB) *del anidamiento más interno se asocia con el primer* `if` ó `If` (C# ó VB) *que encuentra, siguiendo esta asociación desde adentro hacia afuera*

La única forma de alterar esta regla es asociando las sentencias a bloques. Esto provoca que se evalúe el bloque como una unidad del `if` ó `If` (C# ó VB) al que este asociado.

Las sentencias pueden anidarse, de manera de combinar varios condicionales unos dentro de otros.

```
if (expresión booleana)
{
      [sentencias;]
      if (expresión booleana) sentencia;
      if (expresión booleana)
      {
            if (expresión booleana)
            {
                  [sentencias;]
            }
            else
            {
                  [sentencias;]
            }
      }
      else
      {
            [sentencias;]
      }
}
else
{
      [sentencias;]
}
```

Ejemplo

C#

```
public void IfAnidado(int a, int b, int c)
{
    if (a == b)
        if (b == c)
        {
            Console.WriteLine("Equilátero");
            return;
        }
        else
        {
            Console.WriteLine("Isósceles");
            return;
        }
    else
    {
        if (b == c)
        {
            Console.WriteLine("Isósceles");
            return;
        }
        else
        {
```

```
                    if (a == c)
                    {
                        Console.WriteLine("Isósceles");
                        return;
                    }
                    else
                    {
                        Console.WriteLine("Escaleno");
                    }
                }
            }
        }
```

VB

```
    Public Sub IfAnidado(a As Integer, b As Integer, c As Integer)
        If (a = b) Then
            If (b = c) Then
                Console.WriteLine("Equilátero")
                Return
            Else
                Console.WriteLine("Isósceles")
                Return
            End If
        Else
            If (b = c) Then
                Console.WriteLine("Isósceles")
                Return
            Else
                If (a = c) Then
                    Console.WriteLine("Isósceles")
                    Return
                Else
                    Console.WriteLine("Escaleno")
                End If
            End If
        End If
    End Sub
```

Sentencia `switch` de C#

La sintaxis de la sentencia `switch` es la siguiente:

```
switch ( expr1) {
        case constante2:
                [sentencias;]
                <break;> ó < goto case [etiqueta del case]>
        case constante3:
                [sentencias;]
                <break;> ó < goto case [etiqueta del case]>
        default:
                [sentencias;]
                <break;>
}
```

La sentencia `switch` se utiliza para ejecutar sentencias en base al valor que arroja una expresión. Por ejemplo, si un programa contiene un entero llamado mes cuyo valor indica el mes en alguna fecha y se quiere mostrar el nombre del mes basándose en su número entero equivalente, se podría utilizar la sentencia `switch` para realizar esta tarea.

Ejemplo

```
public void ImprimeMes(int mes)
{
    switch (mes)
    {
        case 1: Console.WriteLine("Enero"); break;
        case 2: Console.WriteLine("Febrero"); break;
        case 3: Console.WriteLine("Marzo"); break;
        case 4: Console.WriteLine("Abril"); break;
        case 5: Console.WriteLine("Mayo"); break;
        case 6: Console.WriteLine("Junio"); break;
        case 7: Console.WriteLine("Julio"); break;
        case 8: Console.WriteLine("Agosto"); break;
        case 9: Console.WriteLine("Septiembre"); break;
        case 10: Console.WriteLine("Octubre"); break;
        case 11: Console.WriteLine("Noviembre"); break;
        case 12: Console.WriteLine("Diciembre"); break;
    }
}
```

La sentencia `switch` evalúa la expresión, en este caso el valor de mes, y ejecuta la sentencia `case` apropiada. Cada sentencia case debe ser única y el valor proporcionado a cada sentencia `case` deberá cumplir dos condiciones: ser un valor constante y del mismo tipo de dato que el devuelto por la expresión. Dicho valor puede ser un entero, una cadena o un booleano.

Otro punto interesante son las sentencias `break` después de cada `case`. La sentencia `break` hace que el control salga de la sentencia `switch` y continúe con la siguiente línea. La sentencia `break` es necesaria para indicar la finalización de las sentencias `case`. Esto es, sin un `break` explícito, el compilador marcará un error indicando que no se puede ir del fin de un bloque a otro.

Además, puede utilizar la sentencia `default` al final de la sentencia `switch` para manejar los valores que no se han manejado explícitamente por una de las sentencias `case`. Este bloque también necesita una indicación de finalización como el `case` para que no de error.

Existen dos alternativas a la sentencia `break` para finalizar bloques de `case` en un `switch`: ir desde un case a otro, lo cual se indica con la sentencia `goto case` [etiqueta del case] y `return` que retorna de la función en la que se invoca terminando la ejecución del `switch`. La primera ocasiona que el control de flujo de programa vaya desde un caso a otro y la segunda determina la finalización del método que contiene al `switch`. Si se utiliza con criterio, la primera puede resultar en ahorro de código, sino, puede convertirlo en el algo difícil de mantener a lo largo del tiempo.

Ejemplo

```
public void ArmaAuto1(int modeloDeAuto)
{
    Auto a = new Auto();
```

```
            switch (modeloDeAuto)
            {
                case Auto.DE_LUJO:
                    a.agregarAireAcondicionado();
                    a.agregarRadio();
                    a.agregarRuedas();
                    a.agregarMotor();
                    break;
                case Auto.ESTANDARD:
                    a.agregarRadio();
                    a.agregarRuedas();
                    a.agregarMotor();
                    break;
                default:
                    a.agregarRuedas();
                    a.agregarMotor();
                    break;
            }
        }

        public void ArmaAuto2(int modeloDeAuto)
        {
            Auto a = new Auto();
            switch (modeloDeAuto)
            {
                case Auto.DE_LUJO:
                    a.agregarAireAcondicionado();
                    goto case Auto.ESTANDARD;
                case Auto.ESTANDARD:
                    a.agregarRadio();
                    a.agregarRuedas();
                    a.agregarMotor();
                    break;
            }
        }
```

Hay ciertos escenarios en los que es preferible que el control proceda secuencialmente a través de las sentencias case. Esto se puede hacer sin incluir la sentencia `goto case` siempre y cuando no haya sentencias entre los case que se agrupan. En el siguiente ejemplo, se calcula el número de días de un mes.

Ejemplo

```
        public int DiasDelMes(int mes, int anio)
        {
            int numeroDias=0;

            switch (mes)
            {
                case 1:
                case 3:
                case 5:
                case 7:
                case 8:
                case 10:
```

```
            case 12:
                numeroDias = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numeroDias = 30;
                break;
            case 2:
                if (((anio % 4 == 0) && !(anio % 100 == 0)) || anio % 400 == 0)
                    numeroDias = 29;
                else
                    numeroDias = 28;
                break;
        }
        return numeroDias;
    }
```

El siguiente método Main muestra el uso de las funciones definidas anteriormente

```
    static void Main(string[] args)
    {
        VerificaSeleccionador v = new VerificaSeleccionador();
        Console.WriteLine("Armando auto estándar con el método ArmaAuto1");
        v.ArmaAuto1(Auto.ESTANDARD);
        Console.WriteLine("Armando auto estándar con el método ArmaAuto2");
        v.ArmaAuto2(Auto.ESTANDARD);
        Console.WriteLine("Armando auto de lujo con el método ArmaAuto1");
        v.ArmaAuto1(Auto.DE_LUJO);
        Console.WriteLine("Armando auto de lujo con el método ArmaAuto2");
        v.ArmaAuto2(Auto.DE_LUJO);

        Console.WriteLine("Imprimiendo meses");
        Meses m = new Meses();
        for (int i = 1; i < 13; i++)
        {
            Console.Write("Mes: ");
            m.ImprimeMes(i);
            Console.Write("\tDías del mes :" + m.DiasDelMes(i, 2012).ToString()
                + "\n");
        }
        Console.ReadKey();
    }
```


La salida es

```
Agregando Motor...
Armando auto de lujo con el método ArmaAuto1
Agregando Aire Acondicionado...
Agregando Radio...
Agregando Ruedas...
```

```
Agregando Motor...
Armando auto de lujo con el método ArmaAuto2
Agregando Aire Acondicionado...
Agregando Radio...
Agregando Ruedas...
Agregando Motor...
Imprimiendo meses
Mes: Enero      Días del mes: 31
Mes: Febrero    Días del mes: 29
Mes: Marzo      Días del mes: 31
Mes: Abril      Días del mes: 30
Mes: Mayo       Días del mes: 31
Mes: Junio      Días del mes: 30
Mes: Julio      Días del mes: 31
Mes: Agosto     Días del mes: 31
Mes: Septiembre Días del mes: 30
Mes: Octubre    Días del mes: 31
Mes: Noviembre  Días del mes: 30
Mes: Diciembre  Días del mes: 31
```

El Select Case de VB

Ejecuta uno de varios grupos de instrucciones, según el valor de una expresión. El formato es el siguiente:

```
Select [ Case ] expresión de verificación
  [ Case lista de expresiones
       [sentencias]]
    [ Case Else
       [sentencias del else]]
End Select
```

Donde:

- **Expresión de verificación**: Obligatorio. Es una expresión. Debe evaluarse en uno de los tipos de datos elementales (`Boolean`, `Byte`, `Char`, `Date`, `Double`, `Decimal`, `Integer`, `Long`, `Object`, `Single`, `Short` y `String`).
- **Lista de expresiones**: Se requiere con una instrucción `Case`. Lista de las cláusulas de expresiones que representan valores que coinciden para la **expresión de verificación**. Pueden ser cláusulas con varias expresiones separadas por comas. Cada cláusula puede tomar una de las siguientes formas:
  - expresión1 `To` expresión2
  - [`Is`] operador de comparación expresión

- expresión

Usar la palabra clave `To` para especificar los límites de un rango de valores que coinciden para la **expresión de verificación**. El valor de **expresión1** debe ser menor o igual que el valor de **expresión2**.

Usar la palabra clave `Is` con un operador de comparación (=, <>, <, <=,>, o> =) para especificar una restricción en los valores coincidentes para la **expresión de verificación**. Si la palabra clave `Is` no se proporciona, se inserta automáticamente antes de operador de comparación.

La manera especificando sólo expresión se trata como un caso especial de la forma `Is` donde operador de comparación es el signo igual (=). Esta forma se evalúa como expresión = expresión de verificación

Las expresiones en **lista de expresiones** pueden ser de cualquier tipo de datos, siempre que sean implícitamente convertible al tipo de expresión de verificación y es válido el operador de comparación que se está utilizando es apropiado para los dos tipos.

- **Sentencias**: Opcional. Una o más instrucciones a continuación del `Case` que se ejecutan si expresión de verificación coincide con cualquier cláusula en lista de expresiones.
- **Sentencias del else**: Opcional. Una o más instrucciones a continuación `Case Else` que se ejecutan si expresión de verificación no coincide con ninguna cláusula en lista de expresiones de ninguna de las declaraciones de los `Case`.
- `End Select`: Termina el bloque `Select ... Case`.

Si la expresión de verificación coincide con cualquier cláusula en lista de expresiones de un `Case`, se ejecutan las instrucciones declaradas a continuación de dicho `Case` hasta encontrar la siguiente sentencia `Case` o la instrucción `End Select`. Después, el control pasa a la instrucción que sigue a `End Select`. Si la expresión de verificación coincide con más de una lista de expresiones en las cláusulas `Case`, sólo las declaraciones de instrucciones después de la primera coincidencia se ejecutan.

La instrucción `Case Else` se utiliza para ejecutar las sentencias del else si no hay coincidencia entre la expresión de verificación y ninguna lista de expresiones de ninguna de las declaraciones `Case`. Aunque no es obligatorio, es una buena idea tener una instrucción `Case Else` en el bloque `Select Case` para controlar valores imprevistos en expresión de verificación. Si no hay ninguna cláusula `Case` que posea una lista de expresiones que coincida con la expresión de verificación y no hay ninguna instrucción `Case Else`, la ejecución continúa en la instrucción que sigue a `End Select`.

Se pueden utilizar varias expresiones o intervalos en cada cláusula `Case`. Por ejemplo, la línea siguiente es válida:

```
Case 1 To 4, 7 To 9, 11, 13, Is > MaxNumber
```

Nota: La palabra clave `Is` utilizada en las sentencias `Case` y `Case Else` no es el mismo que el operador de comparación `Is`.

También puede especificar los rangos y expresiones múltiples para cadenas de caracteres. En el siguiente ejemplo, la sentencia coincide con cadenas que son exactamente iguales a las "manzanas", cadenas con valores entre "nueces" y "sopa" en orden alfabético, y el valor actual de TestItem:

```
Case "manzanas", "nueces" To "sopa", TestItem
```

Ejemplo

```
Public Class SelectCase
    Public Sub SeleccionadorComplejo(numero As Integer)
        ' ...
        Select Case numero    ' Evaluar numero.
            Case 1 To 5   ' numero entre 1 y 5 inclusive.
                Console.WriteLine("Entre 1 y 5")
                ' El siguiente es el único caso que se evalúa como verdadero con 8.
            Case 6, 7, 8   ' numero entre 6 y 8.
                Console.WriteLine("Entre 6 y 8")
            Case 9 To 10   ' numero es 9 o 10.
                Console.WriteLine("Más grande que 8")
            Case Else   ' Otros valores.
                Console.WriteLine("No está entre 1 y 10")
        End Select
    End Sub
End Class
```

La sentencia `for` ó `For`

C#

Al igual que otras sentencias explicadas previamente, el `for` puede ejecutarse sobre una única sentencia o sobre un bloque asociado. La sintaxis del ciclo `for` es:

```
for (expresión inicial; expresión booleana; expresión) sentencia;

for (expresión inicial; expresión booleana; expresión)
{
      [sentencias;]
}
```

Donde:

- **expresión inicial**: es la sentencia que se ejecuta una vez al iniciar el ciclo. Se suele utilizar para inicializar variables
- **expresión booleana**: es una sentencia que determina cuando se termina el ciclo. Esta expresión se evalúa al principio de cada iteración en el ciclo. Cuando la expresión se evalúa a `false` el ciclo se termina.

- **Expresión:** se invoca en cada interacción del ciclo a partir del segundo ciclo (inclusive) en adelante.

Cualquiera (o todos) de estos componentes pueden ser una sentencia vacía (un punto y coma)

Ejemplo

```
for (int i = 1; i < 13; i++)
{
       Console.Write("Mes: ");
       m.ImprimeMes(i);
       Console.Write("\tDías del mes: " + m.DiasDelMes(i, 2012).ToString() + "\n");
}
```

Por convención la llave abierta '{' se coloca al final de la misma línea donde se encuentra la sentencia `for` y la llave cerrada '}' empieza una nueva línea con sangría (indentación) en la línea en la que se encuentra el `for`.

VB

La sintaxis de la sentencia es

```
For contador [As Tipo]=valor inicial To valor final [Step incremento]
       [sentencias]
Next
```

Donde

- **Contador**: Se requiere en la instrucción `For`. Variable numérica. La variable de control para el ciclo. Para obtener más información, consulte el contra-argumento más adelante en este tema.
- **Tipo**: Opcional. Tipo de datos del contador. Se requiere cuando se declara el tipo de valor del contador
- **Valor inicial**: Obligatorio. Expresión numérica. El valor inicial del contador.
- **Valor final**: Obligatorio. Expresión numérica. El valor final del contador.
- **Incremento**: Opcional. Expresión numérica. La cantidad en la que contador se incrementa cada vez que inicia un nuevo ciclo.
- **Sentencias**: Opcional. Una o más instrucciones entre el `For` y el `Next` que se ejecutan en el número especificado de ciclos.
- `Next`: Obligatorio. Termina la definición del ciclo `For`.

Cuando un bucle `For` ... `Next` comienza, Visual Basic evalúa valor inicial, valor final e incremento. Este es el único momento en el que se evalúan estos valores. A continuación, asigna valor inicial al contador. Antes de que se ejecute el bloque de instrucciones, compara contador con el valor final. Si el contador es mayor que el valor final (o menor si el incremento es negativo), el ciclo finaliza y el control pasa a la instrucción que sigue a `Next`. De lo contrario el bloque de instrucciones se ejecuta.

Cada vez que Visual Basic encuentra la instrucción `Next`, incrementa contador con incremento y vuelve a la instrucción `For`. De nuevo se compara contador con valor final y de nuevo, o bien

ejecuta el bloque o sale del ciclo, en función del resultado. Este proceso continúa hasta que el contador supere valor final.

Ejemplo

```
Public Sub ForConPaso()
    For number As Double = 2 To 0 Step -0.25
        Console.Write(number.ToString & " ")
    Next
    Console.WriteLine("")
End Sub
```

Sentencia `while` de C# y las sentencias `While`, `Do While` de VB

C#

Una sentencia `while` realiza un ciclo mientras se cumpla una cierta condición (que la expresión que evalúa a un `bool` sea verdadera). La sintaxis general de la sentencia `while` es:

```
while (expresión booleana) sentencia;

while (expresión booleana) {
      [sentencias;]
      }
```

Ejemplo

```
while (i < 10) i++;
```

Al igual que en el `for`, en este ciclo se puede ejecutar una sola sentencia o asociarlo a un bloque.

Ejemplo

```
int i = 0;
while (i < 10)
{
      Console.WriteLine("Iterando …");
      i++;
}
Console.WriteLine("Fin");
```

Por convención la llave abierta '{' se coloca en una nueva línea respecto de donde se encuentra la sentencia `while` y la llave cerrada '}' empieza una nueva línea con sangría (indentación) en la línea en la que se encuentra el `while`.

VB

Una sentencia `While` realiza un ciclo mientras se cumpla una cierta condición (que la expresión que evalúa a un `Boolean` sea verdadera). La sintaxis general de la sentencia `While` es:

```
While expresión booleana
      [sentencias]
End While
```

Ejemplo

```
Public Class SentenciaWhile
    Public Sub WhileSolo()
        Dim counter As Integer = 0
        While counter < 20
            counter += 1
        End While
        Console.WriteLine("El ciclo While se ejecutó " & CStr(counter) & " veces")
    End Sub
End Class
```

Salida:

El ciclo While se ejecutó 20 veces

La sentencia Do While – Loop

VB

Esta sentencia tiene un comportamiento similar al `While`. Su sintaxis es:

```
Do While expresión booleana
      [sentencias]
Loop
```

Ejemplo

```
    Public Sub DoWhileLoop()
        Dim index As Integer = 0
        Do While index <= 10
            Console.Write(index.ToString & " ")
            index += 1
        Loop
        Console.WriteLine("")
    End Sub
```

Sentencia `do-while` de C# y `Do-Loop While` de VB

C#

La sentencia `do-while` se utiliza cuando el ciclo debe ejecutarse al menos una vez. Al final de cada iteración, el ciclo `do` evalúa una expresión booleana. Si esta expresión es verdadera, otra iteración comienza. Si es falsa, el bucle se termina.

La sintaxis del ciclo `do - while` es:

```
do {
      [sentencias;]
} while (expresión booleana) ;
```

Ejemplo

```
int i = 0;
do
{
```

```
            Console.WriteLine("Iterando…");
            i++;
        } while (i < 10);
        Console.WriteLine("Fin");
```

VB

La sentencia `Do-Loop While` se utiliza cuando el ciclo debe ejecutarse al menos una vez. Al final de cada iteración, el ciclo `do` evalúa una expresión booleana. Si esta expresión es verdadera, otra iteración comienza. Si es falsa, el bucle se termina.

La sintaxis del ciclo `Do-Loop While` es:

```
Do
     [sentencias]
Loop While expresión booleana
```

Ejemplo

```
Public Sub DoLoopWhile()
    Dim contador As Integer = 0
    Dim numero As Integer = 5
    Do
        numero = numero - 1
        contador = contador + 1
    Loop While numero > 6
    Console.WriteLine("El Do-Loop While se ejecutó " & contador & " vez.")
End Sub
```

Las sentencias `Do`, `Loop` y `Until`

VB

Hay dos formas de usar la palabra clave `Until` para comprobar una condición en un ciclo `Do`. Se puede comprobar el estado antes de entrar en el ciclo, o se puede comprobar después de que la iteración se ejecute al menos una vez. Los ciclos continúan mientras la condición sea falsa.

El formato de ambas posibilidades es el siguiente

```
Do Until expresión booleana
      [sentencias]
Loop

Do
      [sentencias]
Loop Until expresión booleana
```

En el siguiente ejemplo, el procedimiento VerificarUntilPrimero comprueba la condición antes de entrar en el ciclo. Si numero hubiera sido inicializado en 15 en lugar de 20, las instrucciones dentro del ciclo nunca se ejecutarían. En el procedimiento VerificarUntilAlFinal, las instrucciones dentro del ciclo se ejecutan una vez antes de comprobar la condición, lo cual es falso en la primera prueba.

Ejemplo

```
Sub VerificarUntilPrimero()
    Dim contador As Integer = 0
    Dim numero As Integer = 20
    Do Until numero = 15
        numero = numero - 1
        contador = contador + 1
    Loop
    MsgBox("El Do Until-Loop se ejecutó " & contador & " vez.")
End Sub

Sub VerificarUntilAlFinal()
    Dim contador As Integer = 0
    Dim numero As Integer = 20
    Do
        numero = numero - 1
        contador = contador + 1
    Loop Until numero = 15
    MsgBox("El Do - Loop Until se ejecutó " & contador & " vez.")
End Sub
```

Casos Especiales de Control del Flujo

C#

Sentencias `break` y `continue` en los Ciclos

Como se mostró en la sentencia `switch`, la sentencia `break` hace que el control del flujo salte a la sentencia siguiente respecto de la actual en ejecución.

El caso más simple de `break` interrumpe la sentencia en ejecución. Si se ejecuta dentro de un ciclo que tiene asociado un bloque, como por ejemplo

```
do
{
        sentencia;
        if (expresión booleana)
        {
            break;
        }
        sentencia;
} while (expresión booleana);
Sentencia_fuera_de_bloque;
```

No ejecutará ninguna sentencia más dentro del bloque ni ningún otro ciclo, derivando el flujo del programa a Sentencia_fuera_de_bloque;

Ejemplo

```
public void SentenciaBreak()
{
    for (int i = 0; i < 10; i++)
    {
        Console.Write("\t" + i);
```

```
            if (i == 4) break;
        }

        Console.Write("\n");

        int j = 0;
        while (j < 10)
        {
            j++;
            Console.Write("\t" + j);
            if (j == 5) break;
        }
        Console.WriteLine("\nTerminé!!");
    }
```

La salida es:

```
        0       1       2       3       4
        1       2       3       4       5
Terminé!!
```

La sentencia `continue` es similar pero a diferencia de `break`, hace que se ejecute el próximo ciclo de una iteración. Ambas sentencias interrumpen el flujo normal de la secuencia de sentencias.

El caso de `continue` es forzar a un nuevo ciclo cuando se ejecuta ignorando el resto de las sentencias que se encuentran en ese ciclo posteriores a ella. Si por ejemplo se tiene el siguiente formato:

```
do
{
        sentencia;
    if (expresión booleana) {
            continue;
    }
    sentencia;
} while (expresión booleana);
```

Si se ejecuta el `continue`, la sentencia que se encuentra a continuación del `if` no se ejecutará en ese ciclo.

Ejemplo

```
    public void SentenciaContinue()
    {
        for (int i = 0; i < 10; i++)
        {
            if (i % 2 != 0)
                continue;
            Console.Write("\t" + i);
        }

        Console.Write("\n");
```

```
            int j = 0;
            while (j < 10)
            {
                j++;
                if (j % 2 == 0)
                    continue;
                Console.Write("\t" + j);
            }
            Console.WriteLine("\nTerminé!!");
        }
```

La salida es

```
        0       2       4       6       8
        1       3       5       7       9
Terminé!!
```

Una forma bastante compleja de imprimir números pares e impares que puede sustituirse con código mucho más simple.

Se debe tener cuidado con el uso de estas sentencias. Por ejemplo si se pide a un programador que modifique el código anterior para que sólo se ejecute hasta el número 5, puede decidir modificarlo como se muestra a continuación, empeorando más aún la mala programación.

Ejemplo

```
            for (int i = 0; i < 10; i++)
            {
                if (i % 2 != 0)
                    continue;
                Console.Write("\t" + i);
                if (i == 4) break;
            }

            Console.Write("\n");

            int j = 0;
            while (j < 10)
            {
                j++;
                if (j % 2 == 0)
                    continue;
                Console.Write("\t" + j);
                if (j == 5) break;
            }
            Console.WriteLine("\nTerminé!!");
        }
```

La salida es

```
        0       2       4
        1       3       5
```

```
Terminé!!
```

VB

Sentencias Exit y Continue

La sentencia `Exit` sale de un procedimiento o bloque y transfiere el control inmediatamente a la instrucción que sigue a la llamada de procedimiento o la definición de bloque.

La sintaxis es la siguiente:

```
Exit [<Do | For | Function | Property | Select | Sub | Try | While >]
```

La sentencia en combinación con una instrucción que define un bloque determina la finalización de dicho bloque. Donde

- `Do`: Sale inmediatamente del ciclo `Do` en el que aparece. La ejecución continúa con la instrucción que sigue a la instrucción `Loop`. `Exit Do` sólo se puede utilizar dentro de un bucle `Do`. Cuando se utiliza dentro de ciclos `Do` anidados, `Exit Do` sale del ciclo más interno y transfiere el control al siguiente nivel de anidamiento.
- `For`: Sale inmediatamente del bucle `For` en el que aparece. La ejecución continúa con la instrucción que sigue a `Next`. `Exit For` sólo se puede utilizar dentro de un ciclo `For` ... `Next` o `For Each` ... `Next`. Cuando se utiliza dentro de bucles `For` anidados, `Exit For` sale del ciclo más interno y transfiere el control al siguiente nivel de anidamiento.
- `Function`: Inmediatamente sale del procedimiento `Function` en el que aparece. La ejecución continúa con la instrucción que sigue a la instrucción que llamó al procedimiento `Function`. `Exit Function` sólo se puede utilizar dentro de un procedimiento `Function`.
- `Property`: Inmediatamente sale del procedimiento `Property` en el que aparece. La ejecución continúa con la instrucción que llamó al procedimiento `Property`, es decir, con la sentencia que solicitó o estableció el valor de la propiedad. `Exit Property` sólo se puede utilizar dentro de un procedimiento de propiedad `Get` o `Set`.
- `Select`: Inmediatamente sale del bloque `Select Case` en el que aparece. La ejecución continúa con la instrucción que sigue a `End Select`. `Exit Select` sólo se puede utilizar dentro de una instrucción `Select Case`.
- `Sub`: Sale inmediatamente del procedimiento `Sub` en el que aparece. La ejecución continúa con la instrucción que sigue a la que llamó al procedimiento `Sub`. `Exit Sub` sólo se puede utilizar dentro de un procedimiento `Sub`.
- `Try`: Sale inmediatamente del bloque `Try` o `Catch` en el que aparece. La ejecución **continúa con el bloque** `Finally`, **si lo hay, o con la instrucción que sigue a la instrucción** `End Try` de lo contrario. `Exit Try` sólo se puede utilizar dentro de un bloque `Try` o `Catch`, y **no dentro de un bloque** `Finally`.
- `While`: Sale inmediatamente del ciclo `While` en el que aparece. La ejecución continúa con la instrucción que sigue a `End While`. `Exit While` sólo se puede utilizar dentro de un bucle `While`. Cuando se utiliza dentro de ciclos anidados `While`, `Exit While` transfiere el control al ciclo que está anidado un nivel por encima de aquel en el que se invoca a `Exit While`.

Ejemplo

```
        Console.WriteLine("********Exit For*******")
```

```
        For i As Integer = 0 To 9
            Console.Write("  " + i.ToString)
            If i = 4 Then Exit For
        Next
        Console.WriteLine()

        Console.WriteLine("********Exit While*******")
        Dim j As Integer
        While j < 10
            Console.Write("  " + j.ToString)
            j += 1
            If j = 5 Then Exit While
        End While
        Console.ReadKey()
    End Sub
```

La sentencia Continue transfiere el control inmediatamente a la siguiente iteración de un ciclo.

```
Continue [<Do | For | While>]
```

Se puede transferir desde el interior de un `Do`, `For` o `While` a la siguiente iteración de ese ciclo. El control pasa inmediatamente a la sentencia de verificación del ciclo para ver si éste realiza una nueva iteración, que es equivalente a transferir a la sentencia `For` o `While`, o hacia la sentencia `Do` o `Loop`, que contiene la cláusula `Until` o `While`

Se puede utilizar `Continue` en cualquier lugar dentro de un ciclo que permita transferencias. Por ejemplo, si un ciclo está totalmente contenido dentro de un bloque `Try`, un bloque `Catch` o un bloque `Finally`, se puede utilizar `Continue` para transferir fuera del ciclo. Si, por el contrario, la estructura de la instrucción `Try` ... `End Try` está contenida dentro del bucle, no se puede utilizar `Continue` para transferir el control fuera del bloque `Finally`, y se puede utilizar para transferir de un bloque `Try` o `Catch` sólo si transfiere completamente fuera de la estructura de la instrucción `Try` ... `End Try`.

Si se tienen bucles anidados del mismo tipo, por ejemplo un ciclo `Do` dentro de otro, un `Continue` hace saltos a la declaración siguiente de iteración del ciclo más interno que lo contiene. No se puede utilizar `Continue` para pasar a la siguiente iteración de un ciclo externo.

Si se ha anidado ciclos de diferentes tipos, por ejemplo, un bucle `Do` en un ciclo `For`, puede pasar a la siguiente iteración de cualquiera de los ciclos mediante el uso tanto de `Continue Do` o `Continue For`.

Ejemplo

```
        Console.WriteLine("********Continue For*******")
        For i As Integer = 0 To 9
            If i Mod 2 = 0 Then Continue For
            Console.Write("  " + i.ToString)
        Next
        Console.WriteLine()

        Console.WriteLine("********Continue While*******")
        While j < 10
```

```
            Console.Write("   " + j.ToString)
            j += 1
            If j Mod 2 = 0 Then Continue While
        End While
```

Vectores

Declaración

Como en cualquier lenguaje, un vector agrupa datos del mismo tipo bajo un mismo nombre. La salvedad en estos lenguajes es que los datos pueden ser tipos primitivos o referencias.

Se puede pensar en una vector como una secuencia de elementos. Todos sus elementos tienen el mismo tipo. Se puede construir vectores simples que tienen una dimensión (una lista), dos dimensiones (una tabla), tres dimensiones (un cubo), y así sucesivamente. Los vectores tienen las siguientes características:

Cada elemento del vector contiene un valor.

- Los vectores son indexados a partir del cero. El primer elemento de un vector es elemento 0.
- La longitud de un vector es el número total de elementos que puede contener.
- El límite inferior de un vector es el índice de su primer elemento.
- Los vectores pueden ser unidimensionales, multidimensionales, o escalonados.
- El rango de un vector es el número de dimensiones que posee.

Los vectores de un tipo particular sólo pueden contener elementos de ese tipo. Si se necesita manipular un conjunto de objetos o de tipos por valor, se puede utilizar, como se verá posteriormente, uno de los tipos de colección que se definen en el espacio de nombres System.Collections.

Los vectores en si mismos son referenciados. Esto quiere decir que cuando se declara un vector en realidad primero se declara una referencia a éste que debe ser inicializada posteriormente para poder obtener el vector en si mismo. La forma de declarar referencias a vectores es:

C#

```
char[] s;
Punto[] p;
```

VB

```
Dim s(26) As Char
Dim p(10) As Punto
```

Se debe tener en cuenta que declaraciones como las anteriores sólo crea espacio para una referencia.

Para obtener el espacio de almacenamiento de los elementos del vector, este deberá ser solicitado en tiempo de ejecución, ya que un vector es un objeto y debe ser creado con new. La única excepción a esta regla es cuando el vector se crea y se inicializa al mismo tiempo con un bloque de asignación o inicialización, el cual crea dinámicamente el vector a través de los elementos que lo componen inicialmente (este tema se abordará posteriormente)

Creación

C#

Cuando se crea un vector a partir de la referencia a este, se ejecuta el operador new. Por ejemplo, para un vector de un tipo primitivo (char) el siguiente código muestra la forma de realizarlo:

```
public char[] CrearVector()
{
    char[] s;
    s = new char[26];
    for (int i = 0; i < 26; i++)
    {
        s[i] = (char)('A' + i);
    }
    return s;
}
```

La declaración

```
char[] s;
```

Reserva un espacio en memoria para la referencia s. Para ser más preciso, reserva un espacio en el stack para la referencia, pero todavía no existe un espacio de almacenamiento para los elementos del vector. Cuando se ejecuta la sentencia

```
s = new char[26];
```

VB

Si bien VB no necesita crear exclusivamente un vector a través del operador New, su manejo interno es el mismo y se almacena en el heap.

```
Public Function CrearVector() As Char()
       Dim s(26) As Char
       For i As Integer = 0 To 25
              s(i) = Chr(65 + i)
       Next
       Return s
End Function
```

La declaración

```
Dim s() As Char = New Char(25) {}
```

Reserva un espacio en memoria para la referencia s. Para ser más preciso, reserva un espacio en el stack para la referencia, pero todavía no existe un espacio de almacenamiento para los elementos del vector. Cuando se ejecuta el operador New se reserva el espacio de almacenamiento en el heap

Se reserva dinámicamente en memoria el espacio de almacenamiento para los 26 elementos del vector. El siguiente ejemplo muestra como se realiza la creación y asignación de elementos de un vector (el detalle de como se asignan valores se dará más adelante en este capítulo). Notar que el programa también muestra como pasarlo como parámetro o retornarlo como valor.

Ejemplo

C#

```
namespace vectores
{
    class VectoresDeTiposPrimitivos
    {
        public char[] CrearVector()
        {
            char[] s;

            s = new char[26];
            for (int i = 0; i < 26; i++)
            {
                s[i] = (char)('A' + i);
            }

            return s;
        }

        public void ImprimirVector(char[] vector)
```

```
        {
            Console.Write('<');
            for (int i = 0; i < vector.Length; i++)
            {
                // imprimir un elemento
                Console.Write(vector[i]);
                // Imprimir una coma para delimitarlos
                // si no es el último elemento
                if ((i + 1) < vector.Length)
                {
                    Console.Write(", ");
                }
            }
            Console.Write('>');
        }
    }
}
```

VB

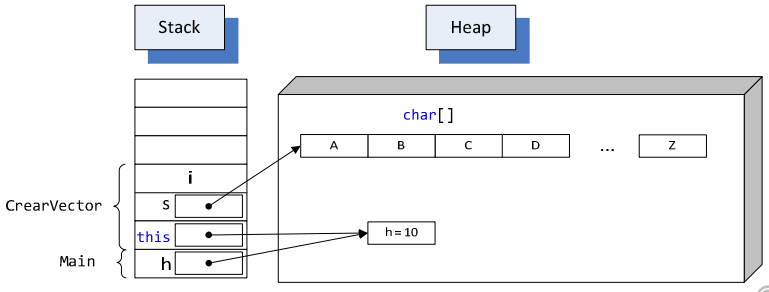
```
Public Class VectoresDeTiposPrimitivos
     Public Function CrearVector() As Char()
        Dim s(26) As Char
        For i As Integer = 0 To 25
            s(i) = Chr(65 + i)
        Next
        Return s
    End Function

    Public Sub ImprimirVector(ByVal vector As Char())
        Console.Write("<")
        For i As Integer = 0 To vector.Length - 1
            ' imprimir un elemento
            Console.Write(vector(i))
            ' imprimir una coma para separar si no es el último elemento
            If (i + 1) < vector.Length Then
                Console.Write(", ")
            End If
        Next
        Console.Write(">")
    End Sub
End Class
```


La salida del programa es

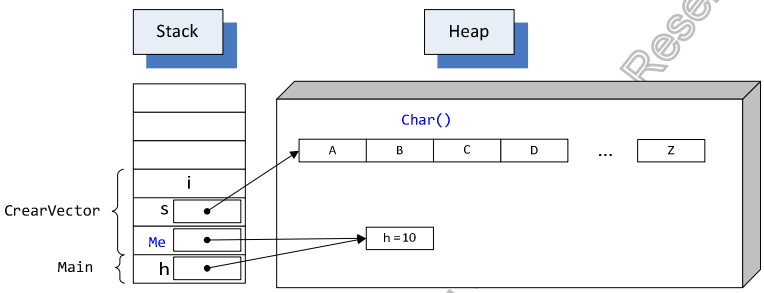
```
<A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z>
```

Cuando un programa se encuentra en ejecución, el único lugar donde se puede requerir memoria dinámicamente es en el heap, por lo tanto los espacios de almacenamiento quedan como muestra la figura.

C#



VB



El hecho de tener que pedir memoria dinámicamente en un vector tiene consecuencias directas cuando se necesita crear un vector de objetos.

En este caso, cuando se declara el espacio para los elementos del vector, cada uno de estos es a la vez una referencia a cada tipo de objeto a crear, por lo tanto, se deberá ejecutar un `new` o `New` (C# o VB) por cada elemento. En el código se utiliza la clase Punto.

Ejemplo

C#

```
namespace vectores
{
    class VectoresDeObjetos
    {
        public Punto[] CrearVector()
        {
            Punto[] p;
            p = new Punto[10];
            for (int i = 0; i < 10; i++)
                p[i] = new Punto(i, i + 1);
            return p;
        }

        public void ImprimirVector(Punto[] vector)
        {
            for (int i = 0; i < vector.Length; i++)
            {
```

```
                // imprimir un elemento
                Console.WriteLine(vector[i].Imprimir());
            }
        }
    }
}
```

VB

```
Public Class VectoresDeObjetos
    Public Function CrearVector() As Punto()
        Dim p(10) As Punto
        For i As Integer = 0 To 9
            p(i) = New Punto(i, i + 1)
        Next
        Return p
    End Function

    Public Sub ImprimirVector(ByVal vector As Punto())
        For i As Integer = 0 To vector.Length - 2
            ' imprimir un elemento
            Console.WriteLine(vector(i).Imprimir())
        Next
    End Sub
End Class
```

La salida del programa es

```
[0,1]
[1,2]
[2,3]
[3,4]
[4,5]
[5,6]
[6,7]
[7,8]
[8,9]
[9,10]
```

Por lo tanto, la declaración

C#

```
Punto[] p;
```

VB

```
Dim p(10) As Punto
```

En C#, sólo crea una referencia en el stack a un vector cuyos elementos serán a la vez referencias de objetos del tipo Punto. La sentencia

```
p = new Punto[10];
```

Crea el espacio de almacenamiento para 10 referencias.

En VB esto se hace todo en la misma declaración

El ciclo a continuación ejecuta la creación de cada elemento por el operador `new` o `New` (C# o VB)

C#

```
for (int i = 0; i < 10; i++)
        p[i] = new Punto(i, i + 1);
```

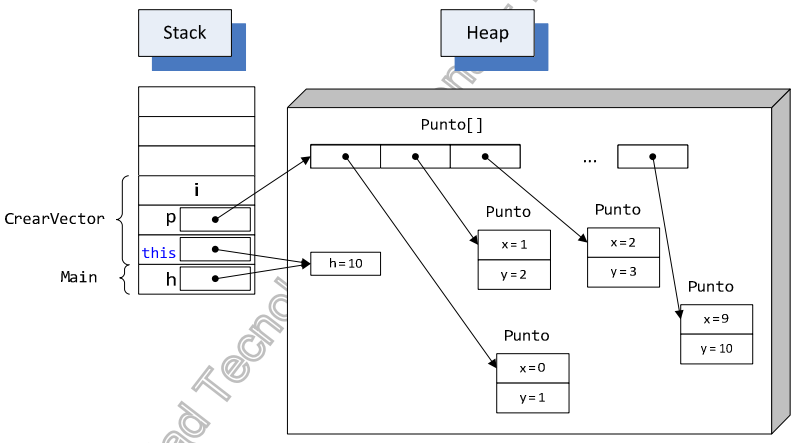
VB

```
For i As Integer = 0 To 9
    p(i) = New Punto(i, i + 1)
Next
```

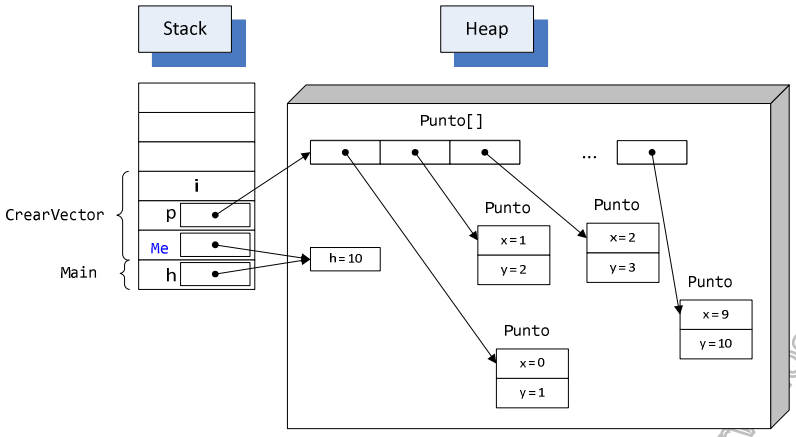
Por lo tanto, es en este momento cuando cada elemento del vector p se va convirtiendo en un objeto.

La siguiente figura es un diagrama de lo expuesto:

C#



VB



Inicialización

Existen dos formas de inicializar los vectores: elemento a elemento (como se mostró anteriormente) o por un bloque de inicialización.

Cuando se inicializa elemento a elemento se necesita crear primero el espacio para almacenar los elementos del vector para luego asignar cada uno de ellos, tanto para los tipos primitivos como para los tipos referenciados. Por ejemplo, para crear e inicializar un vector de elementos primitivos se puede codificar lo siguiente

C#

```
char[] s;
s = new char[4];
s[0] = 'A';
s[1] = 'B';
s[2] = 'C';
s[3] = 'D';
```

VB

```
Dim s(10) As Char
s(0) = "A"
s(1) = "B"
s(2) = "C"
s(3) = "D"
```

Sin embargo, los elementos de un vector de referencia necesitan un poco más de trabajo, ya que se debe crear en cada elemento el objeto acerca del cual se debe almacenar su referencia como un elemento del vector

C#

```
Punto[] p = new Punto[4];
p[0] = new Punto(0, 1);
p[1] = new Punto(1, 2);
p[2] = new Punto(2, 3);
p[3] = new Punto(3, 4);
```

VB

```
Dim p(10) As Punto
p(0) = New Punto(0, 1)
p(1) = New Punto(1, 2)
p(2) = New Punto(2, 3)
p(3) = New Punto(3, 4)
```

Cuando se inicializa un vector con un bloque de asignación, no se ve claramente que el espacio de almacenamiento se asigna como se lo hace con un objeto. Sin embargo el proceso es el mismo pero en lugar de declararlo explícitamente, el lenguaje lo hace automáticamente. Para el caso de tipos primitivos el código es el que se muestra a continuación. Notar el punto y coma luego del bloque. Esto es lo que diferencia un bloque de asignación de uno de declaración

C#

```
int[] vec2={2, 3, 4, 5, 6, 7};
```

VB

```
Dim vec2() As Integer = {2, 3, 4, 5, 6, 7}
```

Nuevamente, los tipos referenciados exigen el esfuerzo adicional de hacer un `new` para cada elemento de manera de crear así el objeto que será referenciado por éste

C#

```
VectoresDeUnaDimension[] vecObj2={new VectoresDeUnaDimension(5,6),
                                  new VectoresDeUnaDimension(7,8),
                                  new VectoresDeUnaDimension(9,10),
                                  new VectoresDeUnaDimension(11,12)};
```

VB

```
Dim vecObj2() As VectoresDeUnaDimension = {
            New VectoresDeUnaDimension(5, 6),
            New VectoresDeUnaDimension(7, 8),
            New VectoresDeUnaDimension(9, 10),
            New VectoresDeUnaDimension(11, 12)
        }
```

Matrices o Vectores Bidimensionales

El concepto de la declaración de la referencia se mantiene en estos casos, con la salvedad que en lugar de un par de corchetes para C# o paréntesis para VB, se utilizan (considerando dos dimensiones) dos, de manera que el primer par especifica la primera dimensión y el otro la segunda.

Existe una alternativa en las declaraciones de varias dimensiones que permite no tener que especificar cada dimensión con un par de corchetes y es separando a las mismas con comas. Sin embargo existe una sutil diferencia entre ambas formas declarativas. Cuando se utilizan corchetes, tomando como ejemplo un vector de dos dimensiones, se considera un ***vector de vectores***, lo que permite la creación de vectores multidimensionales escalonados. Cuando se utiliza la coma ***es una***

matriz cuadrada, no un vector de vectores. Esto implica que no se puede utilizar la propiedad Length para determinar el tamaño de una dimensión en base a otra, sino que se usa para toda la matriz en conjunto.

Ejemplo

CS

```
namespace vectores
{
    class Inicializaciones
    {
        public void VectoresDeVariasDimensiones()
        {
            int[,] mat = new int[,] {
                    {0, 1, 2},
                    {1, 2, 3},
                    {3, 4, 5},
                    {6, 7, 8}
            };
            for (int j = 0; j < 4; j++)
            {
                Console.Write('<');
                for (int i = 0; i < 3; i++)
                {
                    // imprime un elemento
                    Console.Write("[" + mat[j, i] + "]");
                }
                Console.Write('>');
                Console.Write("\n");
            }
        }
    }
}
```

VB

```
Public Class Inicializaciones
    Public Sub VectoresDeVariasDimensiones()

        Dim mat(,) As Integer = New Integer(,) {
                {0, 1, 2},
                {1, 2, 3},
                {3, 4, 5},
                {6, 7, 8}
        }
        For j As Integer = 0 To 3
            Console.Write("<")
            For i As Integer = 0 To 2
                ' imprime un elemento
                Console.Write("[" + mat(j, i).ToString + "]")
            Next i
            Console.Write(">")
            Console.WriteLine()
        Next j
```

```
    End Sub
End Class
```

Los vectores de dos dimensiones o matrices deben pensarse como un vector de vectores cuando se utilizan los corchetes en la declaración en C# o los paréntesis seguidos de un par de llaves en VB, de manera que cada elemento del primer vector mantiene una referencia al primer elemento del vector de la segunda dimensión, por lo tanto, cualquier intento de inicializar la segunda dimensión antes que la primera será un error en tiempo de compilación. Un ejemplo de las posibles declaraciones válidas y el error descripto se muestra a continuación

C#

```
int[][] dosDim = new int[4][];
dosDim[0] = new int[5];
dosDim[1] = new int[5];
int[][] mat = new int [][4]; // Error
```

VB

```
Dim dosDim()() As Integer = New Integer(3)() {}
Dim mal() As Integer = New Integer()()() {} ' Error
Dim temp1(5) As Integer
dosDim(0) = temp1
Dim temp2(5) As Integer
dosDim(1) = temp2
```

Las matrices no necesitan ser cuadradas en su creación. Recordando el concepto de considerarlas vectores de vectores, cada vector de la segunda dimensión se puede crear con los elementos que se necesiten. Por ejemplo

C#

```
dosDim[0] = new int[2];
dosDim[1] = new int[4];
dosDim[2] = new int[6];
dosDim[3] = new int[8];
```

VB

```
Dim dosDim()() As Integer = New Integer(3)() {}
Dim mal() As Integer = New Integer()() {} ' Error
Dim temp1(5) As Integer
dosDim(0) = temp1
Dim temp2(8) As Integer
dosDim(1) = temp2
```

Límite de un Vector

Lo siguiente deseable al declarar e inicializar un vector, tenga una o varias dimensiones, es recorrerlo. Para ello los principales temas a tener en cuenta son dónde empieza y hasta dónde llega. El primer tema es fácil, ya que todos los vectores comienzan por el elemento cuyo subíndice es 0. El segundo tema no es más difícil, puesto que los lenguajes brindan la posibilidad de acceder

a una propiedad que contiene la cantidad de elementos que el vector tiene la capacidad de almacenar.

Luego, si todos los elementos del vector poseen valores iniciales, recorrerlo se limita a establecer los límites superiores e inferiores y usar un ciclo de iteración. Se debe tener especial cuidado cuando los elementos del vector sean referencias a objetos, que los mismo guarden una referencia válida a uno de ellos antes de utilizarlo o generará un error en tiempo de ejecución.

Un ejemplo de recorrido de un vector, que sólo contiene los valores iniciales que le dan los lenguajes a las variables de tipo entero en cada elemento, es el siguiente:

C#

```
for(int i=0; i < vec.Length; i++)
        Console.WriteLine("Elemento " + i +":" + vec[i]);
```

VB

```
For i As Integer = 0 To vec.Length -1
        Console.WriteLine("Elemento " + i +":" + vec(i))
Next
```


Recorriendo un vector con `foreach` ó `For Each`

La instrucción `foreach` ó `For Each` (C# ó VB) repite un grupo de instrucciones incluidas en el ciclo para cada elemento de un vector o de un objeto del tipo Collection (lo cual se verá posteriormente). La instrucción `foreach` ó `For Each` (C# ó VB) se utiliza para recorrer en iteración una colección de elementos y obtener la información deseada, pero no se debe utilizar para cambiar el contenido de la colección, ya que se pueden producir efectos secundarios imprevisibles.

Las instrucciones del ciclo siguen ejecutándose para cada elemento del vector, la matriz o colección. Cuando ya se han recorrido todos los elementos, el control se transfiere a la siguiente instrucción fuera del bloque `foreach` ó `For Each` (C# ó VB).

En cualquier punto dentro del bloque `foreach` ó `For Each` (C# ó VB), se puede salir del bucle utilizando la palabra clave `break` ó `Exit For` (C# ó VB), o pasando directamente la iteración siguiente del bucle mediante la palabra clave `continue` ó `Continue` (C# ó VB).

También se puede salir de un bucle `foreach` ó `For Each` (C# ó VB) mediante las instrucciones `goto`, `return` o `throw` y `Goto`, `Return` o `Throw` (C# ó VB).

Esta instrucción proporciona un modo simple y limpio de recorrer en iteración los elementos de un vector. Por ejemplo, el siguiente código recorre de esta manera un vector de caracteres:

Ejemplo

C#

```
namespace vectores
{
```

```
    class ForEachConVector
    {
        public char[] CrearVector()
        {
            char[] s;
            s = new char[26];
            for (int i = 0; i < 26; i++)
            {
                s[i] = (char)('A' + i);
            }
            return s;
        }

        public void ImprimirVector(char[] vector)
        {
            int i = 0;
            Console.Write('<');
            foreach (char elemento in vector)
            {
                // imprimir un elemento
                Console.Write(elemento);
                // imprimir una coma para separar si no es el último elemento
                if ((i + 1) < vector.Length)
                {
                    Console.Write(", ");
                }
                i++;
            }
            Console.Write('>');
        }
    }
}
```

VB

```
Public Class ForEachConVector
    Public Function CrearVector() As Char()
        Dim s(26) As Char
        For i As Integer = 0 To 25
            s(i) = Chr(65 + i)
        Next
        Return s
    End Function

    Public Sub ImprimirVector(ByVal vector As Char())
        Dim i As Integer = 0
        Console.Write("<")
        For Each elemento As Char In vector
            ' imprimir un elemento
            Console.Write(elemento)
            ' imprimir una coma para separar si no es el último elemento
            If (i + 1) < vector.Length Then
                Console.Write(", ")
                i += 1
            End If
        Next
```

```
        Console.Write(">")
    End Sub
End Class
```

Redimensionamiento de un Vector

Nota: VB incluye la instrucción `Redim` para redimensionar un vector sin conservar sus valores y `Redim Preserve` para redimensionar conservándolos. Se mostrará en lugar de estas instrucciones Resize para unificar criterios con C#

Para redimensionar un vector se utiliza Resize. Este método aloja un nuevo vector con el tamaño especificado, los elementos del vector de origen se copian al nuevo, para terminar remplazando el antiguo vector con el nuevo. Si el vector es null, este método crea uno nuevo con el tamaño especificado.

Ejemplo

C#

```
void Array.Resize(ref <Vector>, int <Nuevo tamaño>);
```

VB

```
void Array.Resize(<Vector>, ByVal <Nuevo tamaño> As Integer)
```

Si el nuevo tamaño especificado es mayor que la longitud del vector de origen, uno nuevo se alojará y todos los elementos se copiarán del vector de origen al nuevo. En cambio si el tamaño es menor que la longitud del vector de origen, se aloja un nuevo vector y se copian elementos del origen al nuevo hasta que el nuevo se llena y el resto de los elementos ignoran. Si tamaño especificado es igual a la longitud del vector de origen, este método no hace nada.

Ejemplo

CS

```
namespace vectores
{
    class Redimensionamientos
    {
        public void Redimensionar()
        {
            // Crear e inicializar un nuevo vector de cadenas.
            String[] miVector = {"Esta", "es", "una", "historia", "corta",
            "de", "viejos", "y", "queridos", "amigos"};

            // Mostrar los valores almacenados en el vector.
            Console.WriteLine(
                "El vector de cadenas continene incialmente los " +
              "siguientes valores:");
            ImprimirIndicesYValores(miVector);
```

```
            // Redimensionar el vector a un tamaño mayor (cinco elementos más).
            Array.Resize(ref miVector, miVector.Length + 5);

            // Imprimir en pantalla los valores contenidos actualmente.
            Console.WriteLine("Luego de redimensionar a un tamaño mayor, ");
            Console.WriteLine("La cadena contiene los siguientes valores:");
            ImprimirIndicesYValores(miVector);

            // Redimensionar el vector a un tamaño menor (dejar cuatro
              // elementos solamente).
            Array.Resize(ref miVector, 4);

            // Imprimir en pantalla los valores contenidos actualmente.
            Console.WriteLine("Luego de redimensionar a un tamaño menor, ");
            Console.WriteLine("La cadena contiene los siguientes valores:");
            ImprimirIndicesYValores(miVector);
        }

        public void ImprimirIndicesYValores(String[] miVector)
        {
            for (int i = 0; i < miVector.Length; i++)
            {
                Console.WriteLine("   [{0}] : {1}", i, miVector[i]);
            }
            Console.WriteLine();
        }
    }
}
```

VB

```
Public Class Redimensionamientos
    Public Sub Redimensionar()
        ' Crear e inicializar un nuevo vector de cadenas.
        Dim miVector As String() = {"Esta", "es", "una", "historia", "corta", _
            "de", "viejos", "y", "queridos", "amigos"}

        ' Mostrar los valores almacenados en el vector.
        Console.WriteLine( _
            "El vector de cadenas continene incialmente los siguientes valores:")
        ImprimirIndicesYValores(miVector)

        ' Redimensionar el vector a un tamaño mayor (cinco elementos más).
        Array.Resize(miVector, miVector.Length + 5)

        ' Imprimir en pantalla los valores contenidos actualmente.
        Console.WriteLine("Luego de redimensionar a un tamaño mayor, ")
        Console.WriteLine("La cadena contiene los siguientes valores:")
        ImprimirIndicesYValores(miVector)

        ' Redimensionar el vector a un tamaño mayor (cinco elementos más).
        Array.Resize(miVector, 4)

        ' mprimir en pantalla los valores contenidos actualmente.
        Console.WriteLine("Luego de redimensionar a un tamaño mayor, ")
        Console.WriteLine("La cadena contiene los siguientes valores:")
```

```
        ImprimirIndicesYValores(miVector)
    End Sub

    Public Shared Sub ImprimirIndicesYValores(miVector() As String)
        Dim i As Integer
        For i = 0 To miVector.Length - 1
            Console.WriteLine("   [{0}] : {1}", i, miVector(i))
        Next i
        Console.WriteLine()
    End Sub
End Class
```

Copia de Vectores

Para copiar vectores se pueden copiar los elementos uno a uno o utilizar el método Copy. Éste copia un intervalo de elementos de un vector de origen comenzando en el índice de origen especificado y los almacena en otro vector comenzando en el índice de destino especificado. La longitud y los índices se especifican como enteros de 32 bits.

Ejemplo

```
Array.Copy(<origen>, <índice del origen>, <destino>,
                <índice del destino>, <longitud>)
```

Los parámetros origen y destino deben tener el mismo número de dimensiones.

Cuando se realizan operaciones de copia entre matrices multidimensionales, el vector se comporta como un gran vector unidimensional, en el que las filas (o columnas) se distribuyen conceptualmente de un extremo a otro. Por ejemplo, si un vector tiene tres filas (o columnas) con cuatro elementos cada una, una operación de copia de seis elementos desde el principio del mismo, copiará los cuatro elementos de la primera fila (o columna) y los primeros dos elementos de la segunda fila (o columna). Para empezar a copiar a partir del segundo elemento de la tercera fila (o columna), el índice del origen debe ser la suma del límite superior de la primera fila (o columna) y la longitud de la segunda fila (o columna) más dos.

Si el vector de origen y el vector de destino se superponen, este método se comporta como si los valores originales del origen se preservaran en una ubicación temporal antes de que se sobrescriba el de destino.

Los vectores pueden ser tipos por referencia o tipo por valor. Si es necesario, se realiza una conversión de tipo.

- Cuando se realiza una operación de copia de un vector con tipos por referencia a uno con tipos por valor, a cada elemento se le aplica una conversión unboxing y, a continuación, se copia. Cuando se realiza una operación de copia que es a la inversa de la anterior, a cada elemento se le aplica una conversión boxing y, a continuación, se copia.

- Cuando se realiza una operación de copia de un vector de tipos por referencia o de tipos por valor a un vector del tipo Object, se crea un Object donde se almacena cada valor o referencia y, a continuación, se copia. Cuando se realiza una operación de copia de un vector de Object a un vector de tipo de valor o de referencia y la asignación no es posible, se produce una excepción InvalidCastException (las excepciones se tratarán posteriormente).
- Si el origen y el destino son vectores de tipo por referencia o de tipo Object, se realiza una copia superficial. Una copia superficial de un vector es uno nuevo que contiene referencias a los mismos elementos que el vector original. No se copian los propios elementos ni ningún otro objeto al que hagan referencia. Por otro lado, una copia profunda de vector, copia los elementos y todo aquello a lo que hacen referencia directa o indirectamente dichos elementos.

Si los vectores son de tipos incompatibles, se produce una excepción ArrayTypeMismatchException. La compatibilidad de tipos se define a continuación:

Si este método produce una excepción durante la operación de copia, el estado del el vector de destino queda sin definir.

Ejemplo

C#

```
namespace vectores
{
    class CopiaDeVectores
    {
        public void Copiar()
        {
            // Crear e inicializar un nuevo vector de Int32.
            Int32[] vectorDeInt32Original = new Int32[5] ;
            for (int i = 0; i < vectorDeInt32Original.Length; i++)
                vectorDeInt32Original[i] = i + 1;

            // Crear e inicializar un nuevo vector de Int32.
            Int32[] otroVectorDeInt32 = new Int32[5];
            for (int i = 0; i < otroVectorDeInt32.Length; i++)
                otroVectorDeInt32[i] = i + 26;

            // Imprimir en pantalla los valores contenidos en ambos vectores.
            Console.WriteLine("Vector de Int32:");
            ImprimeVector(vectorDeInt32Original);
            Console.WriteLine("Otro Vector de Int32:");
            ImprimeVector(otroVectorDeInt32);

            // Copiar el primer elemento del vector de Int32 al del otro vector.
            Array.Copy(vectorDeInt32Original, 0, otroVectorDeInt32, 0, 1);
            // Copiar los últimos dos elementos del otro vector al de Int32
              // original.
```

```
            Array.Copy(otroVectorDeInt32, otroVectorDeInt32.Length - 2,
              vectorDeInt32Original, vectorDeInt32Original.Length - 2, 2);
            // Mostrar por pantalla los valores modificados de ambos vectores.
            Console.WriteLine("Vector de Int32 original - Los últimos dos " +
              "elementos son iguales al del otro vector de Int32:");
            ImprimeVector(vectorDeInt32Original);
            Console.WriteLine("Otro vector de Int32 - El primer elemento es " +
              "igual al del vector de Int32 original:");
            ImprimeVector(otroVectorDeInt32);
        }

        private void ImprimeVector(Int32[] vec)
        {
            Console.Write('<');
            for (int i = 0; i < vec.Length; i++)
            {
                // imprime un elemento
                Console.Write(vec[i]);
                // imprime una coma para delimitar si no es el último elemento
                if ((i + 1) < vec.Length)
                {
                    Console.Write(", ");
                }
            }
            Console.Write('>');
            Console.Write("\n");
        }
    }
}
```

VB

```
Public Class CopiaDeVectores
    Public Sub Copiar()

        ' Crear e inicializar un nuevo vector de Int32.
        Dim vectorDeInt32Original() As Int32 = New Int32(5) {}
        Dim i As Integer
        For i = 0 To vectorDeInt32Original.Length - 1
            vectorDeInt32Original(i) = i + 1
        Next
        ' Crear e inicializar un nuevo vector de Object.
        Dim otroVectorDeInt32() As Int32 = New Int32(5) {}
        For i = 0 To otroVectorDeInt32.Length - 1
            otroVectorDeInt32(i) = i + 26
        Next

        ' Imprimir en pantalla los valores contenidos en ambos vectores.
        Console.WriteLine("Vector de Int32:")
        ImprimeVector(vectorDeInt32Original)
        Console.WriteLine("Otro Vector de Int32:")
        ImprimeVector(otroVectorDeInt32)

        ' Copiar el primer elemento del vector de Int32 al del otro vector.
        Array.Copy(vectorDeInt32Original, 0, otroVectorDeInt32, 0, 1)
        ' Copiar los últimos dos elementos del otro vector al de Int32 original.
```

```
        Array.Copy(otroVectorDeInt32, otroVectorDeInt32.Length - 2, _
              vectorDeInt32Original, vectorDeInt32Original.Length - 2, 2)
        ' Mostrar por pantalla los valores modificados de ambos vectores.
        Console.WriteLine("Vector de Int32 original - Los últimos dos " +
                          "elementos son iguales al del otro vector de Int32:")
        ImprimeVector(vectorDeInt32Original)
        Console.WriteLine("Otro vector de Int32 - El primer elemento es igual " +
                          "al del vector de Int32 original:")
        ImprimeVector(otroVectorDeInt32)
    End Sub

    Private Sub ImprimeVector(vec() As Int32)
        Dim i As Integer
        Console.Write("<")
        For i = 0 To vec.Length - 1

            ' imprime un elemento
            Console.Write(vec(i))
            ' imprime una coma para delimitar si no es el último elemento
            If ((i + 1) < vec.Length) Then
                Console.Write(", ")
            End If
        Next
        Console.Write(">")
        Console.WriteLine()
    End Sub
End Class
```