INSTRUCCIONES EN C# .NET 2005

switch

La instrucción **switch** es una instrucción de control que controla múltiples selecciones y enumeraciones pasando el control a una de las instrucciones **case** de su cuerpo, como se muestra en el ejemplo siguiente:

```
int caseSwitch = 1;
switch (caseSwitch)
{
   case 1:
      Console.WriteLine("Case 1");
      break;
   case 2:
      Console.WriteLine("Case 2");
      break;
   default:
      Console.WriteLine("Default case");
      break;
}
```

El control se transfiere a la instrucción **case** que coincide con el valor del modificador. La instrucción **switch** puede incluir cualquier número de instancias **case**, sin embargo dos instrucciones case nunca pueden tener el mismo valor. La ejecución del cuerpo de la instrucción empieza en la instrucción seleccionada y continúa hasta que la instrucción **break** transfiere el control fuera del cuerpo **case**. Es necesario introducir una instrucción de salto como **break** después de cada bloque **case**, incluido el último bloque, se trate de una instrucción **case** o de una instrucción **default**. Con una excepción, (a diferencia de la instrucción **switch** de C++), C# no admite el paso implícito de una etiqueta case a otra. Esta excepción se produce si una instrucción **case** no tiene ningún código.

Si ninguna expresión case coincide con el valor de la instrucción switch, entonces el control se transfiere a las instrucciones que siguen la etiqueta **default** opcional. Si no existe ninguna etiqueta **default**, el control se transfiere fuera de la instrucción **switch**.

continue

La instrucción **continue** transfiere el control a la siguiente iteración de la instrucción de iteración envolvente donde aparece.

En este ejemplo, se inicializa un contador que cuenta de 1 a 10. Utilizando la instrucción **continue** con la expresión (i < 9), se omiten las instrucciones situadas entre **continue** y el final del cuerpo del bucle for.

```
// statements_continue.cs
using System;
class ContinueTest
{
   static void Main()
   {
      for (int i = 1; i <= 10; i++)
      {
         if (i < 9)
         {
            continue;
         }
```

Resultados:

9
10

```
            Console.WriteLine(i);
        }
    }
}
```

do

La instrucción **do** ejecuta una instrucción o un bloque de instrucciones entre **{}** repetidamente hasta que una expresión especificada se evalúe como **false**. En el ejemplo siguiente las instrucciones de bucle **do** se ejecutan con la condición de que la variable `y` sea menor que 5.

```
// statements_do.cs
using System;
public class TestDoWhile
{
   public static void Main ()
   {
      int x = 0;
      do
      {
         Console.WriteLine(x);
         x++;
      }
      while (x < 5);
   }
}
```

Resultados:

```
0
1
2
3
4
```

A diferencia de la instrucción while, el cuerpo del bucle de la instrucción **do** se ejecuta al menos una vez, independientemente del valor de la expresión.

while

La instrucción **while** ejecuta una instrucción o un bloque de instrucciones repetidamente hasta que una expresión especificada se evalúa como **false**.

```
// statements_while.cs
using System;
class WhileTest
{
   static void Main()
   {
      int n = 1;
      while (n < 6)
      {
         Console.WriteLine("El valor actual de n es {0}", n);
         n++;
      }
   }
}
```

Resultados:
El valor actual de n es 1
El valor actual de n es 2
El valor actual de n es 3
El valor actual de n es 4
El valor actual de n es 5

Otra forma:

```
// statements_while_2.cs
using System;
class WhileTest
{
   static void Main()
   {
      int n = 1;
      while (n++ < 6)
      {
         Console.WriteLine("El valor actual de n es {0}", n);
      }
   }
}
```

Resultados:
El valor actual de n es 1
El valor actual de n es 2
El valor actual de n es 3
El valor actual de n es 4
El valor actual de n es 5

for

El bucle **for** ejecuta una instrucción o un bloque de instrucciones repetidamente hasta que una determinada expresión se evalúa como **false**. El bucle **for** es útil para recorrer en iteración matrices y para procesar secuencialmente. En el ejemplo siguiente el valor de int i se escribe en la consola y el valor de i se va incrementando en 1 en el bucle.

```
// statements_for.cs
// for loop
using System;
class ForLoopTest
{
   static void Main()
   {
      for (int i = 1; i <= 5; i++)
      {
         Console.WriteLine(i);
      }
   }
}
```

Resultados:
1
2
3
4
5

La instrucción **for** ejecuta la instrucción o instrucciones internas repetidamente del siguiente modo:

- Primero, se evalúa el valor inicial de la variable i.
- A continuación, mientras el valor de i es menor que 5, la condición se evalúa como **true**, se ejecuta la instrucción Console.WriteLine y se reevalúa i.
- Cuando i es mayor que 5, la condición se convierte en **false** y el control se transfiere fuera del bucle.

Puesto que la comprobación de la expresión condicional tiene lugar antes de la ejecución del bucle, las instrucciones internas de un bucle **for** pueden no llegar a ejecutarse.

Todas las expresiones de la instrucción **for** son opcionales; por ejemplo, la siguiente instrucción se utiliza para crear un bucle infinito:

```
      for (;;)
{
   // ...
}
```

foreach, in

La instrucción **foreach** repite un grupo de instrucciones incluidas en el bucle para cada elemento de una matriz o de un objeto collection. La instrucción **foreach** 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 bucle siguen ejecutándose para cada elemento de la matriz o la colección. Cuando ya se han recorrido todos los elementos de la colección, el control se transfiere a la siguiente instrucción fuera del bloque **foreach**.

*En este ejemplo, **foreach** se utiliza para mostrar el contenido de una matriz de enteros.*

```
// cs_foreach.cs
class ForEachTest
{
   static void Main(string[] args)
   {
      int[] fibarray = new int[] { 0, 1, 2, 3, 5, 8, 13 };
      foreach (int i in fibarray)
      {
         System.Console.WriteLine(i);
      }
   }
}
```

```
Resultados:
0
1
2
3
5
8
13
```

if-else

La instrucción **if** selecciona una instrucción para ejecución en base al valor de una expresión **Boolean**. En el ejemplo siguiente un indicador **Boolean** flagCheck se establece en **true** y, a continuación, se protege en la instrucción **if**. El resultado es: The flag is set to true.

```
bool flagCheck = true;
if (flagCheck == true)
{
   Console.WriteLine("El flag está establecido para true.");
}
else
{
   Console.WriteLine("El flag está establecido para false.");
}
```

Si la expresión en el paréntesis se evalúa como true, a continuación se ejecuta la instrucción Console.WriteLine("El flag boolean es establecido para true."); Después de ejecutar la instrucción **if**, el control se transfiere a la siguiente instrucción. Else no se ejecuta en este ejemplo.

Si se desea ejecutar más de una instrucción, es posible ejecutar varias instrucciones en forma condicional al incluirlas en bloques mediante **{}**, al igual que en el ejemplo anterior.

Las instrucciones que se van a ejecutar como resultado de comprobar la condición pueden ser de cualquier tipo, incluida otra instrucción **if** anidada dentro de la instrucción **if** original. En las instrucciones **if** anidadas, la cláusula **else** pertenece a la última instrucción **if** que no tiene una cláusula **else** correspondiente. Por ejemplo:

```
      if (x > 10)
   if (y > 20)
      Console.Write("Declaracion_1");
   else
```

```
    Console.Write("Declaracion_2");
```

En este ejemplo, se mostrará **Declaracion_1** si la condición (y > 20) se evalúa como **false**. No obstante, si desea asociar **Declaracion_2** a la condición (x >10), utilice llaves:

```
    if (x > 10)
{
  if (y > 20)
    Console.Write("Declaracion_1");
}
else
  Console.Write("Declaracion_2");
```

En este caso, se mostrará **Declaracion_2** si la condición (x > 10) se evalúa como **false**.

En este ejemplo, se escribe un carácter desde el teclado y el programa comprueba si se trata de un carácter alfabético. En ese caso, comprueba si es minúscula o mayúscula. En cada caso, se muestra el mensaje apropiado.

```
// statements_if_else.cs
// if-else example
using System;
class IfTest
{
  static void Main()
  {
    Console.Write("Ingrese un carácter: ");
    char c = (char)Console.Read();
    if (Char.IsLetter(c))
    {
      if (Char.IsLower(c))
      {
        Console.WriteLine("El carácter es minúscula.");
      }
      else
      {
        Console.WriteLine("El carácter es mayúscula.");
      }
    }
    else
    {
      Console.WriteLine("No es un carácter alfabético.");
    }
  }
}
```

Resultado del ejemplo:
Ingrese un carácter: 2
No es un carácter alfabético.

A continuación se ofrece otro ejemplo:
Ejecución Nº 2:
Ingrese un carácter: A
El carácter es mayúscula.

Ejecución Nº 3:
Ingrese un carácter: h
El carácter es minuscula.

También es posible extender la instrucción **if** de modo que puedan controlarse varias condiciones, mediante la construcción else-if siguiente:

```
        if (condicion_1)
{
   // Declaracion_1;
}
else if (condicion_2)
{
   // Declaracion_2;
}
else if (condicion_3)
{
   // Declaracion_3;
}
else
{
   // Declaracion_n;
}
```

Este ejemplo comprueba si el carácter especificado es una letra minúscula, mayúscula o un número. En cualquier otro caso, se tratará de un carácter no alfanumérico. El programa utiliza la anterior estructura else-if en escalera.

```
// statements_if_else2.cs
// else-if
using System;
public class IfTest
{
   static void Main()
   {
      Console.Write("Ingrese un carácter: ");
      char c = (char)Console.Read();

      if (Char.IsUpper(c))
      {
         Console.WriteLine("El carácter es mayúscula.");
      }
      else if (Char.IsLower(c))
      {
         Console.WriteLine("El carácter es minúscula.");
      }
      else if (Char.IsDigit(c))
      {
         Console.WriteLine("El carácter es un numero.");
      }
      else
      {
         Console.WriteLine("El carácter no es un alfanumérico.");
      }
   }
}
```

Resultados del ejemplo
Ingrese un carácter: E
El carácter es mayúscula.

A continuación se ofrecen otros ejemplos de ejecuciones:
Ejecución Nº 2:
Ingrese un carácter: e
El carácter es minúscula.

Ejecución Nº 3:
Ingrese un carácter: 4
El carácter es un número.

Ejecución Nº 4:
Ingrese un carácter: $
El carácter no es un alfanumérico.

goto

La instrucción **goto** transfiere el control del programa directamente a una instrucción identificada por una etiqueta.
Un uso habitual de **goto** consiste en transferir el control a una etiqueta switch-case específica o a la etiqueta predeterminada de una instrucción **switch**.
La instrucción **goto** también es útil para salir de bucles de varios niveles de anidamiento.
*El ejemplo siguiente muestra cómo utilizar **goto** en una instrucción switch.*

```
// statements_goto_switch.cs
using System;
class SwitchTest
{
   static void Main()
   {
      Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
      Console.Write("Please enter your selection: ");
      string s = Console.ReadLine();
      int n = int.Parse(s);
      int cost = 0;
      switch (n)
      {
         case 1:
            cost += 25;
            break;
         case 2:
            cost += 25;
            goto case 1;
         case 3:
            cost += 50;
            goto case 1;
         default:
            Console.WriteLine("Invalid selection.");
            break;
      }
      if (cost != 0)
      {
         Console.WriteLine("Please insert {0} cents.", cost);
      }
      Console.WriteLine("Thank you for your business.");
   }
}
```

Resultados:
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.

*El siguiente ejemplo muestra el uso de **goto** para salir de un conjunto de bucles anidados.*

```
// statements_goto.cs
// Nested search loops
using System;
public class GotoTest1
{
   static void Main()
   {
      int x = 200, y = 4;
      int count = 0;
      string[,] array = new string[x, y];

      // Inicializa el array:
      for (int i = 0; i < x; i++)

         for (int j = 0; j < y; j++)
            array[i, j] = (++count).ToString();

      // Lectura de entrada:
      Console.Write("Ingrese un numero para la búsqueda: ");

      // Input a string:
      string myNumber = Console.ReadLine();

      // Search:
      for (int i = 0; i < x; i++)
      {
         for (int j = 0; j < y; j++)
         {
            if (array[i, j].Equals(myNumber))
            {
               goto Encontrado;
            }
         }
      }

      Console.WriteLine("El numero {0} no fue encontrado.", myNumber);
      goto Finalizar;

   Encontrado:
      Console.WriteLine("El numero {0} fue encontrado.", myNumber);

   Finalizar:
      Console.WriteLine("Finaliza la busqueda.");
   }
}
Resultados:
Enter the number to search for: 44
The number 44 is found.
End of search.
```

is

Comprueba si un objeto es compatible con un tipo determinado. Por ejemplo, se puede determinar si un objeto es compatible con el tipo **string** de la forma siguiente:

```
if (obj is string)
{
}
```

Una expresión **is** se evalúa como **true** si la expresión proporcionada no es NULL y el objeto proporcionado se puede convertir al tipo proporcionado sin producir una excepción.

La palabra clave **is** tiene como resultado una advertencia en tiempo de compilación si se sabe que la expresión siempre será **true** o siempre será **false**, pero normalmente evalúa la compatibilidad de tipos en tiempo de ejecución.

El operador **is** no se puede sobrecargar.

Observe que el operador **is** solamente tiene en cuenta las conversiones de referencia, las conversiones boxing y las conversiones unboxing. No se tienen en cuenta otras conversiones, tales como las conversiones definidas por el usuario.

```
// cs_keyword_is.cs
// The is operator.
using System;
class Class1
{
}
class Class2
{
}

class IsTest
{
   static void Test(object o)
   {
      Class1 a;
      Class2 b;

      if (o is Class1)
      {
         Console.WriteLine("o es Class1");
         a = (Class1)o;
         // Do something with "a."
      }
      else if (o is Class2)
      {
         Console.WriteLine("o es Class2");
         b = (Class2)o;
         // Do something with "b."
      }
      else
      {
         Console.WriteLine("o is neither Class1 nor Class2.");
      }
   }
   static void Main()
   {
```

```
    Class1 c1 = new Class1();
    Class2 c2 = new Class2();
    Test(c1);
    Test(c2);
    Test("a string");
  }
}
```

Resultados:
o es Class1
o es Class2
o is neither Class1 nor Class2.

return

La instrucción **return** termina la ejecución del método en el que aparece y devuelve el control al método que realizó la llamada. También puede devolver un valor opcional. Si el método es del tipo **void**, la instrucción **return** se puede omitir.

En el siguiente ejemplo, el método A() devuelve la variable Area como un valor de tipo double.

```
// statements_return.cs
using System;
class ReturnTest
{
   static double CalculateArea(int r)
   {
      double area = r * r * Math.PI;
      return area;
   }

   static void Main()
   {
      int radius = 5;
      Console.WriteLine("The area is {0:0.00}", CalculateArea(radius));
   }
}
```

Resultado: The area is 78.54

this

La palabra clave **this** hace referencia a la instancia actual de la clase.

A continuación, se indican algunos usos comunes de **this**:

Obtener acceso a miembros con el fin de evitar ambigüedades con nombres similares, por ejemplo:

```
public Employee(string name, string alias)
{
   this.name = name;
   this.alias = alias;
}
```

Pasar un objeto como parámetro a otros métodos, por ejemplo, para:

```
CalcTax(this);
```

Declarar indizadores, por ejemplo:

```
public int this [int param]
{
   get { return array[param];  }
   set { array[param] = value; }
}
```

*Debido a que las funciones miembro estáticas existen en el nivel de clase y no como parte de un objeto, no tienen un puntero **this**. Es un error hacer referencia a **this** en un método estático.*

*En este ejemplo, **this** se utiliza para calificar los miembros de la clase Employee, name y alias, que presentan ambigüedad con nombres similares. También se utiliza para pasar un objeto al método CalcTax, el cual pertenece a otra clase.*

```
// keywords_this.cs
// this example
using System;
class Employee
{
   private string name;
   private string alias;
   private decimal salary = 3000.00m;

   // Constructor:
   public Employee(string name, string alias)
   {
      // Use this to qualify the fields, name and alias:
      this.name = name;
      this.alias = alias;
   }

   // Printing method:
   public void printEmployee()
   {
      Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
      // Passing the object to the CalcTax method by using this:
      Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
   }

   public decimal Salary
   {
      get { return salary; }
   }
}
class Tax
{
   public static decimal CalcTax(Employee E)
   {
      return 0.08m * E.Salary;
   }
}

class MainClass
{
   static void Main()
   {
      // Create objects:
      Employee E1 = new Employee("John M. Trainer", "jtrainer");

      // Display results:
      E1.printEmployee();
   }
}
```

Resultados:

Name: John M. Trainer
Alias: jtrainer
Taxes: $240.00

enum

La palabra clave **enum** se utiliza para declarar una enumeración, un tipo distinto que consiste en un conjunto de constantes con nombre denominado lista de enumeradores. Cada tipo de enumeración tiene un tipo subyacente, que puede ser cualquier tipo integral excepto char. El tipo predeterminado subyacente de los elementos de la enumeración es int. De forma predeterminada, el primer enumerador tiene el valor 0 y el valor de cada enumerador sucesivo se incrementa en 1.
Por ejemplo:

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

*En esta enumeración, Sat es 0, Sun es 1, Mon es 2 y así sucesivamente. Los enumeradores pueden tener inicializadores que reemplazan a los valores predeterminados**. Por ejemplo:***

```
enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

En esta enumeración, se obliga a que la secuencia de elementos empiece en 1 en vez de en 0.

A una variable de tipo Days se le puede asignar cualquier valor en el intervalo del tipo subyacente; los valores no se limitan a las constantes con nombre.

El valor predeterminado de un **enum** E es el valor producido por la expresión (E)0.

Nota: Un enumerador no puede contener espacio en blanco en su nombre.

*El tipo subyacente especifica el almacenamiento asignado para cada enumerador. No obstante, se necesita una conversión explícita para convertir un tipo **enum** a un tipo integral. Por ejemplo, la siguiente instrucción asigna el enumerador Sun a una variable de tipo int utilizando una conversión explícita para convertir de **enum** a int:*

```
int x = (int)Days.Sun;
```

*Cuando se aplica System.FlagsAttribute a una enumeración que contiene algunos elementos combinados con una operación OR bit a bit, se observará que el atributo afecta el comportamiento de **enum** cuando se utiliza con algunas herramientas. Se pueden observar estos cambios al utilizar herramientas tales como los métodos de la clase **Console**, el Evaluador de expresiones, etc.*
Asignar valores adicionales a nuevas versiones de enumeraciones o cambiar los valores de los miembros de enumeración en una nueva versión, puede producir problemas para el código fuente dependiente. Es común que los valores **enum** se utilicen en instrucciones switch y, si se han agregado elementos adicionales al tipo **enum**, la comprobación de los valores predeterminados puede devolver true de forma inesperada.

Si otros desarrolladores van a utilizar su código, es importante proporcionar instrucciones sobre cómo su código debe reaccionar si se agregan nuevos elementos a cualquier tipo **enum**.

En este ejemplo, se declara la enumeración Days. Dos enumeradores se convierten explícitamente en un número entero y se asignan a variables de número entero.

```
// keyword_enum.cs
// enum initialization:
using System;
public class EnumTest
{
   enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};

   static void Main()
   {
      int x = (int)Days.Sun;
      int y = (int)Days.Fri;
```

```
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
```

Resultados:
Sun = 2
Fri = 7

En este ejemplo, la opción de tipo base se utiliza para declarar un **enum** cuyos miembros son del tipo **long**. Observe que a pesar de que el tipo subyacente de la enumeración es **long**, los miembros de la enumeración todavía deben convertirse explícitamente al tipo **long** mediante una conversión de tipos.

```
// keyword_enum2.cs
// Using long enumerators
using System;
public class EnumTest
{
    enum Range :long {Max = 2147483648L, Min = 255L};
    static void Main()
    {
        long x = (long)Range.Max;
        long y = (long)Range.Min;
        Console.WriteLine("Max = {0}", x);
        Console.WriteLine("Min = {0}", y);
    }
}
```

Resultados:
Max = 2147483648
Min = 255

struct

Un tipo **struct** es un tipo de valor que se suele utilizar para encapsular pequeños grupos de variables relacionadas, como las coordenadas de un rectángulo o las características de un elemento de un inventario. En el ejemplo siguiente se muestra una declaración de estructura sencilla.

```
public struct Book
{
    public decimal price;
    public string title;
    public string author;
}
```

Las estructuras también pueden contener constructores, constantes, campos, métodos, propiedades, indizadores, operadores, eventos y tipos anidados, aunque si se requieren estos miembros, se debe considerar la posibilidad de crear una clase en vez de un tipo.

Las estructuras pueden implementar una interfaz, pero no pueden heredar de otra estructura. Por esa razón, los miembros de estructura no se pueden declarar como **protected**.

typeof

Obtenga el objeto **System.Type** para un tipo. Una expresión **typeof** se presenta de la siguiente forma:

```
System.Type type = typeof(int);
```

Para obtener el tipo de una expresión en tiempo de ejecución, puede utilizar el método GetType de .NET Framework de la manera siguiente.

```
int i = 0;
System.Type type = i.GetType();
```

El operador **typeof** también se puede utilizar en tipos de genéricos abiertos. Los tipos con más de un parámetro de tipo deben tener el número adecuado de comas en la especificación. El operador **typeof** no se puede sobrecargar.

En este ejemplo se utiliza el método **GetType** para determinar el tipo utilizado para contener el resultado de un cálculo numérico. Esto depende de los requisitos de almacenamiento del número resultante.

```
// cs_operator_typeof2.cs
using System;
class GetTypeTest
{
   static void Main()
   {
      int radius = 3;
      Console.WriteLine("Area = {0}", radius * radius * Math.PI);
      Console.WriteLine("The type is {0}",
           (radius * radius * Math.PI).GetType());
   }
}
Resultados:
Area = 28.2743338823081
The type is System.Double
```

const

La palabra clave **const** se utiliza para modificar una declaración de un campo o una variable local. Especifica que el valor del campo o de la variable local es constante, o sea que no se puede modificar. Por ejemplo:

```
const int x = 0;
public const double gravitationalConstant = 6.673e-11;
private const string productName = "Visual C#";
```

El tipo de una declaración de constantes especifica el tipo de los miembros que se incluyen en la declaración. Una expresión constante debe dar un valor del tipo destino o de un tipo que se pueda convertir implícitamente en el tipo destino.

Una expresión constante es una expresión que se puede evaluar completamente en tiempo de compilación. Por consiguiente, los únicos valores posibles para constantes de tipos de referencia son **string**, y **null**.

La declaración de constante puede declarar varias constantes, por ejemplo:

```
public const double x = 1.0, y = 2.0, z = 3.0;
```

El modificador **static** no se puede utilizar en una declaración de constante.
Una constante puede participar en una expresión constante, por ejemplo:

```
public const int c1 = 5;
public const int c2 = c1 + 100;
```

Nota: La palabra clave readonly es diferente de la palabra clave **const**. Un campo **const** sólo puede inicializarse en la declaración del campo. Un campo **readonly** puede inicializarse en la declaración o en un constructor. Por lo tanto, los campos **readonly** pueden tener diferentes valores en función del constructor que se utilice. Además, mientras que un campo **const** es una constante en tiempo de compilación, el campo **readonly** puede utilizarse para constantes en tiempo de ejecución, como muestra la línea siguiente: public static readonly uint l1 = (uint)DateTime.Now.Ticks;

```
// const_keyword.cs
using System;
public class ConstTest
{
   class SampleClass
   {
      public int x;
      public int y;
      public const int c1 = 5;
      public const int c2 = c1 + 5;

      public SampleClass(int p1, int p2)
      {
         x = p1;
         y = p2;
      }
   }

   static void Main()
   {
      SampleClass mC = new SampleClass(11, 22);
      Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
      Console.WriteLine("c1 = {0}, c2 = {1}",
                  SampleClass.c1, SampleClass.c2 );
   }
}
```

Resultados:
x = 11, y = 22
c1 = 5, c2 = 10

public

La palabra clave **public** es un modificador de acceso para tipos y miembros de tipos. El acceso de tipo public corresponde al nivel de acceso menos restrictivo. No existen restricciones para obtener acceso a los miembros públicos, como en este ejemplo:

```
class SampleClass
{
   public int x; // ninguna restricción de acceso.
}
```

En el siguiente ejemplo, se declaran dos clases, Point y MainClass. El acceso a los miembros públicos x e y de Point se realiza directamente desde MainClass.

```
// protected_public.cs
// Public access
using System;
class Point
{
   public int x;
   public int y;
}

class MainClass
{
   static void Main()
   {
      Point p = new Point();
      // Direct access to public members:
      p.x = 10;
```

```
      p.y = 15;
      Console.WriteLine("x = {0}, y = {1}", p.x, p.y);
   }
}
```

Resultado:
x = 10, y = 15
Si se cambia el nivel de acceso de **public** a private o protected, se aparecerá el siguiente mensaje de error:

'Point.y' is inaccessible due to its protection level.

static

Utilice el modificador **static** para declarar un miembro estático, que pertenece al propio tipo en vez de a un objeto específico. El modificador **static** puede utilizarse con clases, campos, métodos, propiedades operadores y eventos, pero no puede utilizarse con indizadores, destructores o tipos que no sean clases. Por ejemplo, la siguiente clase se declara como **static** y solo contiene métodos **static**:

```
    static class CompanyEmployee
{
   public static string GetCompanyName(string name) { ... }
   public static string GetCompanyAddress(string address) { ... }
}
```

- Una declaración de constante o tipo constituye, implícitamente, un miembro estático.
- No se puede hacer referencia a un miembro estático por medio de una instancia. En vez de ello, se debe hacer referencia por medio del nombre de tipo. Por ejemplo, considere la siguiente clase:

```
public class MyBaseC
{
   public struct MyStruct
   {
      public static int x = 100;
   }
}
```

- Para referirse al miembro estático x, use el nombre completo (a menos que sea accesible desde el mismo ámbito):

```
MyBaseC.MyStruct.x
```

try

La instrucción try-catch consta de un bloque **try** seguido de una o más cláusulas **catch**, las cuales especifican controladores para diferentes excepciones.
El bloque **try** contiene el código protegido que puede causar la excepción. Este bloque se ejecuta hasta que se produce una excepción o hasta completarse satisfactoriamente. Por ejemplo, el siguiente intento de convertir un objeto **null** provoca la excepción NullReferenceException:

```
object o2 = null;
try
{
   int i2 = (int)o2;   // Error
```

```
}
```
La cláusula **catch** se puede utilizar sin argumentos, en cuyo caso captura cualquier tipo de excepción y se conoce como cláusula catch general. También puede aceptar un argumento de objeto derivado de **System.Exception**, en cuyo caso trata una excepción específica. Por ejemplo:

```
catch (InvalidCastException e)
{
}
```
Es posible utilizar más de una cláusula **catch** específica en la misma instrucción try-catch. En este caso, el orden de las cláusulas **catch** es importante, ya que las cláusulas **catch** se examinan por orden. Las excepciones más específicas se capturan antes que las menos específicas.

Se puede utilizar una instrucción throw en el bloque **catch** para volver a producir la excepción, la cual ha sido capturada por la instrucción **catch**. Por ejemplo:

```
catch (InvalidCastException e)
{
   throw (e);    // Rethrowing exception e
}
```
Si desea volver a producir la excepción que está siendo actualmente controlada por una cláusula **catch** sin parámetros, use la instrucción **throw** sin argumentos. Por ejemplo:

```
catch
{
   throw;
}
```
Dentro de un bloque **try**, inicialice sólo variables declaradas en su interior; en caso contrario, puede producirse una excepción antes de que se complete la ejecución del bloque. Por ejemplo, en el siguiente ejemplo de código, la variable x se inicializa dentro del bloque **try**. Al intentar utilizar esta variable fuera del bloque **try** en la instrucción Write(x), **se generará el siguiente error del compilador: Uso de variable local no asignada.**

```
static void Main()
{
   int x;
   try
   {
      // Don't initialize this variable here.
      x = 123;
   }
   catch
   {
   }
   // Error: Use of unassigned local variable 'x'.
   Console.Write(x);
}
```

Un uso común de **catch** y **finally** consiste en obtener y utilizar recursos en un bloque **try**, tratar circunstancias excepcionales en el bloque **catch** y liberar los recursos en el bloque **finally**.

```
// try_catch_finally.cs
using System;
public class EHClass
{
   static void Main()
   {
      try
      {
```

```
            Console.WriteLine("Ejecutando la sentencia try.");
            throw new NullReferenceException();
        }
        catch (NullReferenceException e)
        {
            Console.WriteLine("{0} Caught exception #1.", e);
        }
        catch
        {
            Console.WriteLine("Caught exception #2.");
        }
        finally
        {
            Console.WriteLine("Bloque finally ejecutado.");
        }
    }
}
```

Resultados del ejemplo

```
Ejecutando la sentencia try.
System.NullReferenceException: Object reference not set to an instance of an object.
   at EHClass.Main() Caught exception #1.
Bloque finally ejecutado.
```

get

Define un método de *descriptor de acceso* en una propiedad o indizador que recupera el valor de la propiedad o el elemento del indizador

Éste es un ejemplo de un descriptor de acceso **get** para una propiedad denominada Seconds:

```
class TimePeriod
{
   private double _seconds;
   public double Seconds
   {
      get { return _seconds;  }
      set { _seconds = value; }
   }
}
```

set

Define un método de *descriptor de acceso* en una propiedad o indizador que estableció el valor de la propiedad o el elemento del indizador.
Éste es un ejemplo de un descriptor de acceso **set** para una propiedad denominada `Seconds`:

```
class TimePeriod
{
   private double _seconds;
   public double Seconds
   {
      get { return _seconds;  }
      set { _seconds = value; }
   }
}
```

Convert

Convierte un tipo de datos base en otro tipo de datos base.

Esta clase devuelve un tipo cuyo valor es equivalente al valor de un tipo especificado. Los tipos base que se admiten son Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime y String.

Existe un método de conversión para convertir todos y cada uno de los tipos base en los demás tipos base. Sin embargo, la operación de conversión real efectuada queda incluida en tres categorías:

- La conversión de un tipo en sí mismo devuelve dicho tipo. No se lleva a cabo realmente ninguna conversión.
- La conversión que no puede producir un resultado significativo produce una excepción InvalidCastException. No se lleva a cabo realmente ninguna conversión. Las conversiones de **Char** en **Boolean**, **Single**, **Double**, **Decimal** o **DateTime**, y de estos tipos en **Char** producen una excepción. Las conversiones de **DateTime** en cualquier tipo excepto **String**, y de cualquier tipo excepto **String** en **DateTime** producen una excepción.
- Los tipos base no descritos anteriormente pueden ser objeto de conversiones a y desde cualquier otro tipo base.

No se producirá una excepción si la conversión de un tipo numérico produce una pérdida de precisión, es decir, la pérdida de algunos de los dígitos menos significativos. Sin embargo, la excepción se producirá si el resultado es mayor de lo que puede representar el tipo de valor devuelto del método de conversión.

Por ejemplo, cuando un tipo **Double** se convierte en un tipo **Single**, se puede producir una pérdida de precisión pero no se produce ninguna excepción. Sin embargo, si la magnitud del tipo **Double** es demasiado grande para que un tipo **Single** lo represente, se produce una excepción de desbordamiento.

Existe un conjunto de métodos que admiten la conversión de una matriz de bytes en y desde un tipo **String** o una matriz de caracteres Unicode formada por dígitos de base 64. Los datos expresados como dígitos de base 64 se pueden transmitir fácilmente en canales de datos que sólo pueden transmitir caracteres de 7 bits.

Algunos de los métodos de esta clase toman un objeto de parámetro que implementa la interfaz IFormatProvider. Este parámetro puede proporcionar información de formato específica de la referencia cultural para ayudar en el proceso de conversión. Los tipos de valor base pasan por alto este parámetro, pero los tipos definidos por el usuario que implementan IConvertible pueden tenerlo en cuenta.

*En el siguiente ejemplo de código, se muestran algunos de los métodos de conversión de la clase **Convert**, entre los que se incluyen ToInt32, ToBoolean y ToString.*

```
double dNumber = 23.15;

try {
   // Returns 23
   int   iNumber = System.Convert.ToInt32(dNumber);
}
catch (System.OverflowException) {
   System.Console.WriteLine(
            "Overflow in double to int conversion.");
}
// Returns True
bool   bNumber = System.Convert.ToBoolean(dNumber);

// Returns "23.15"
string strNumber = System.Convert.ToString(dNumber);

try {
```

```
   // Returns '2'
   char chrNumber = System.Convert.ToChar(strNumber[0]);
}
catch (System.ArgumentNullException) {
   System.Console.WriteLine("String is null");
}
catch (System.FormatException) {
   System.Console.WriteLine("String length is greater than 1.");
}

// System.Console.ReadLine() returns a string and it
// must be converted.
int newInteger = 0;
try {
   System.Console.WriteLine("Ingrese un entero:");
   newInteger = System.Convert.ToInt32(
                System.Console.ReadLine());
}
catch (System.ArgumentNullException) {
   System.Console.WriteLine("la cadena es nula.");
}
catch (System.FormatException) {
   System.Console.WriteLine("String does not consist of an " +
            "optional sign followed by a series of digits.");
}
catch (System.OverflowException) {
   System.Console.WriteLine(
   "Overflow in string to int conversion.");
}

System.Console.WriteLine("Your integer as a double is {0}",
                System.Convert.ToDouble(newInteger));
```