
C#
www.postparaprogramadores.com

Contenido

C # - Herencia

C # - Polimorfismo

C # - Sobrecarga del operador

C # - Interfaces

C # - Espacios de nombres

C # - Directivas de preprocesador

C # - Expresiones regulares

C # - Manejo de excepciones

C # - E / S de archivo

Tutorial avanzado de C #

C # - Atributos

C # - Reflexión

C # - Propiedades

C # - Indizadores

C # - Delegados

C # - Eventos

C # - Colecciones

C # - Genéricos

C # - Métodos anónimos

C # - Códigos inseguros

C # - Multithreading

C# - Descripción general

C # es un lenguaje de programación moderno, de propósito general y orientado a objetos desarrollado por Microsoft y aprobado por la Asociación Europea de Fabricantes de Computadoras (ECMA) y la Organización Internacional de Normalización (ISO).

C # fue desarrollado por Anders Hejlsberg y su equipo durante el desarrollo de .Net Framework.

C # está diseñado para Common Language Infrastructure (CLI), que consiste en el código ejecutable y el entorno de tiempo de ejecución que permite el uso de varios lenguajes de alto nivel en diferentes plataformas y arquitecturas informáticas.

Las siguientes razones hacen de C # un lenguaje profesional ampliamente utilizado:

- Es un lenguaje de programación moderno y de uso general.
- Está orientado a objetos.
- Está orientado a componentes.
- Es facil de aprender.
- Es un lenguaje estructurado.
- Produce programas eficientes.
- Se puede compilar en una variedad de plataformas informáticas.
- Es parte de .Net Framework.

Fuertes características de programación de C

Aunque las construcciones de C # siguen de cerca los lenguajes tradicionales de alto nivel, C y C ++ son un lenguaje de programación orientado a objetos. Tiene una gran semejanza con Java, tiene numerosas características de programación fuertes que lo hacen atractivo para varios programadores de todo el mundo.

La siguiente es la lista de algunas características importantes de C #:

- Condiciones booleanas
- Recolección Automática de Basura
- Biblioteca estándar
- Versiones de ensamblaje
- Propiedades y eventos
- Delegados y Gestión de Eventos
- Genéricos fáciles de usar
- Indexadores
- Compilación condicional
- Multithreading simple
- LINQ y expresiones lambda
- Integración con Windows

C # - Medio ambiente

Pruébelo Opción en línea

Hemos configurado el entorno de programación C # en línea, para que pueda compilar y ejecutar todos los ejemplos disponibles en línea. Le da confianza en lo que está leyendo y le permite verificar los programas con diferentes opciones. Siéntase libre de modificar cualquier ejemplo y ejecutarlo en línea.

Pruebe el siguiente ejemplo utilizando nuestro compilador en línea disponible en CodingGround

```
using System;

namespace HelloWorldApplication {

   class HelloWorld {

      static void Main(string[] args) {
         /* my first program in C# */
         Console.WriteLine("Hello World");
         Console.ReadKey();
      }
   }
}
```

Para la mayoría de los ejemplos dados en este tutorial, encontrará una opción Pruébelo en las secciones de código de nuestro sitio web en la esquina superior derecha que lo llevará al compilador en línea. Así que solo utilízalo y disfruta de tu aprendizaje.

En este capítulo, analizaremos las herramientas necesarias para crear la programación de C #. Ya hemos mencionado que C # es parte de .Net framework y se usa para escribir aplicaciones .Net. Por lo tanto, antes de analizar las herramientas disponibles para ejecutar un programa C #, déjenos comprender cómo C # se relaciona con el marco .Net.

El marco .Net

.Net Framework es una plataforma revolucionaria que le ayuda a escribir los siguientes tipos de aplicaciones:

- Aplicaciones de Windows
- aplicaciones web
- servicios web

Las aplicaciones de .Net framework son aplicaciones multiplataforma. El marco ha sido diseñado de tal manera que puede usarse desde cualquiera de los siguientes lenguajes: C #, C ++, Visual Basic, Jscript, COBOL, etc. Todos estos lenguajes pueden acceder al marco y comunicarse entre sí.

El marco .Net consta de una enorme biblioteca de códigos utilizados por los lenguajes del cliente, como C #. Los siguientes son algunos de los componentes del marco .Net:

- Common Language Runtime (CLR)
- La biblioteca de clases de .Net Framework
- Especificación de lenguaje común
- Sistema de tipo común
- Metadatos y Asambleas
- Windows Forms
- ASP.Net y ASP.Net AJAX
- ADO.Net
- Windows Workflow Foundation (WF)
- Fundación de presentación de Windows
- Windows Communication Foundation (WCF)
- LINQ

Para los trabajos que realiza cada uno de estos componentes, consulte ASP.Net - Introducción , y para obtener detalles de cada componente, consulte la documentación de Microsoft.

Entorno de desarrollo integrado (IDE) para C

Microsoft proporciona las siguientes herramientas de desarrollo para la programación de C #:

- Visual Studio 2010 (VS)
- Visual C # 2010 Express (VCE)
- Visual Web Developer

Los dos últimos están disponibles gratuitamente en el sitio web oficial de Microsoft. Con estas herramientas, puede escribir todo tipo de programas C # desde simples aplicaciones de línea de comandos hasta aplicaciones más complejas. También puede escribir archivos de código fuente de C # utilizando un editor de texto básico, como el Bloc de notas, y compilar el código en ensamblajes utilizando el compilador de línea de comandos, que nuevamente forma parte de .NET Framework.

Visual C # Express y Visual Web Developer Express son versiones recortadas de Visual Studio y tienen la misma apariencia. Conservan la mayoría de las características de Visual Studio. En este tutorial, hemos usado Visual C # 2010 Express.

Puede descargarlo de Microsoft Visual Studio . Se instala automáticamente en su máquina.

Nota: Necesita una conexión a Internet activa para instalar la edición express.

Escribir programas de C # en Linux o Mac OS

Aunque .NET Framework se ejecuta en el sistema operativo Windows, existen algunas versiones alternativas que funcionan en otros sistemas operativos. **Mono** es una versión de código abierto de .NET Framework que incluye un compilador de C # y se ejecuta en varios sistemas operativos, incluidos varios sabores de Linux y Mac OS. Compruebe amablemente Go Mono .

El propósito declarado de Mono no es solo poder ejecutar aplicaciones Microsoft .NET multiplataforma, sino también brindar mejores herramientas de desarrollo para desarrolladores de Linux. Mono se puede ejecutar en muchos sistemas operativos, incluidos Android, BSD, iOS, Linux, OS X, Windows, Solaris y UNIX.

C # - Estructura del programa

Antes de estudiar los componentes básicos del lenguaje de programación C #, veamos una estructura mínima mínima del programa C # para que podamos tomarlo como referencia en los próximos capítulos.

Creando el Programa Hello World

El programa AC # consta de las siguientes partes:

- Declaración de espacio de nombres
- Una clase
- Métodos de clase
- Atributos de clase
- Un método principal
- Declaraciones y expresiones

- Comentarios

Veamos un código simple que imprime las palabras "Hola Mundo" -

```
using System;

namespace HelloWorldApplication {

   class HelloWorld {

      static void Main(string[] args) {
         /* my first program in C# */
         Console.WriteLine("Hello World");
         Console.ReadKey();
      }
   }
}
```

Cuando este código se compila y ejecuta, produce el siguiente resultado:

```
Hello World
```

Veamos las diversas partes del programa dado:

- La primera línea del programa **usando el sistema;** - la palabra clave **using** se usa para incluir el espacio de nombres del **sistema** en el programa. Un programa generalmente tiene múltiples declaraciones de **uso** .
- La siguiente línea tiene la declaración del **espacio de nombres** . Un **espacio de nombres** es una colección de clases. El espacio de nombres *HelloWorldApplication* contiene la clase *HelloWorld* .
- La siguiente línea tiene una declaración de **clase** , la clase *HelloWorld* contiene las definiciones de datos y métodos que utiliza su programa. Las clases generalmente contienen múltiples métodos. Los métodos definen el comportamiento de la clase. Sin embargo, la clase *HelloWorld* solo tiene un método **Main** .
- La siguiente línea define el método **Main** , que es el **punto de entrada** para todos los programas de C #. El método **Main** establece lo que hace la clase cuando se ejecuta.
- El compilador ignora la siguiente línea /*...*/ y agrega **comentarios** en el programa.
- El método Main especifica su comportamiento con la instrucción **Console.WriteLine ("Hello World");**

  WriteLine es un método de la clase de *consola* definida en el espacio de nombres del *sistema* . Esta declaración provoca el mensaje "¡Hola, mundo!" para ser exhibido en la pantalla.
- La última línea **Console.ReadKey ();** es para los usuarios de VS.NET. Esto hace que el programa espere una pulsación de tecla y evita que la pantalla se ejecute y se cierre rápidamente cuando el programa se inicia desde Visual Studio .NET.

Vale la pena señalar los siguientes puntos:

- C # distingue entre mayúsculas y minúsculas.
- Todas las declaraciones y expresiones deben terminar con un punto y coma (;).

- La ejecución del programa comienza en el método Main.
- A diferencia de Java, el nombre del archivo del programa podría ser diferente del nombre de la clase.

Compilar y ejecutar el programa

Si está utilizando Visual Studio.Net para compilar y ejecutar programas de C #, siga estos pasos:

- Inicie Visual Studio.
- En la barra de menú, elija Archivo -> Nuevo -> Proyecto.
- Elija Visual C # de las plantillas y luego elija Windows.
- Elija la aplicación de consola.
- Especifique un nombre para su proyecto y haga clic en el botón Aceptar.
- Esto crea un nuevo proyecto en el Explorador de soluciones.
- Escribir código en el editor de código.
- Haga clic en el botón Ejecutar o presione la tecla F5 para ejecutar el proyecto. Aparece una ventana del símbolo del sistema que contiene la línea Hello World.

Puede compilar un programa de C # utilizando la línea de comandos en lugar del IDE de Visual Studio:

- Abra un editor de texto y agregue el código mencionado anteriormente.
- Guarde el archivo como **helloworld.cs**
- Abra la herramienta de símbolo del sistema y vaya al directorio donde guardó el archivo.
- Escriba **csc helloworld.cs** y presione Intro para compilar su código.
- Si no hay errores en su código, el símbolo del sistema lo lleva a la siguiente línea y genera el archivo ejecutable **helloworld.exe** .
- Escriba **helloworld** para ejecutar su programa.
- Puede ver la salida Hello World impresa en la pantalla.

C # - Sintaxis básica

C # es un lenguaje de programación orientado a objetos. En la metodología de Programación Orientada a Objetos, un programa consta de varios objetos que interactúan entre sí mediante acciones. Las acciones que puede realizar un objeto se denominan métodos. Se dice que los objetos del mismo tipo tienen el mismo tipo o, se dice que están en la misma clase.

Por ejemplo, consideremos un objeto Rectángulo. Tiene atributos como largo y ancho. Dependiendo del diseño, puede necesitar formas de aceptar los valores de estos atributos, calcular el área y mostrar detalles.

Veamos la implementación de una clase Rectangle y analicemos la sintaxis básica de C #:

```
using System;

namespace RectangleApplication {

   class Rectangle {
      // member variables
      double length;
      double width;

      public void Acceptdetails() {
         length = 4.5;
         width = 3.5;
      }

      public double GetArea() {
         return length * width;
      }

      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }

   class ExecuteRectangle {

      static void Main(string[] args) {
         Rectangle r = new Rectangle();
         r.Acceptdetails();
         r.Display();
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

El *uso* palabras clave

La primera declaración en cualquier programa de C # es

```
using System;
```

El **uso** palabra clave se usa para incluir los espacios de nombres en el programa. Un programa puede incluir múltiples declaraciones de uso.

La *clase* palabra clave de

La **clase** palabra clave se usa para declarar una clase.

Comentarios en C

Los comentarios se utilizan para explicar el código. Los compiladores ignoran las entradas de comentarios. Los comentarios de varias líneas en los programas de C # comienzan con / * y terminan con los caracteres * / como se muestra a continuación:

```
/* This program demonstrates
The basic syntax of C# programming
Language */
```

Los comentarios de una sola línea se indican con el símbolo '//'. Por ejemplo,

```
}//end class Rectangle
```

Variables miembro

Las variables son atributos o miembros de datos de una clase, utilizados para almacenar datos. En el programa anterior, la clase *Rectangle* tiene dos variables miembro llamadas *largo* y *ancho* .

Funciones de miembros

Las funciones son un conjunto de declaraciones que realizan una tarea específica. Las funciones miembro de una clase se declaran dentro de la clase. Nuestra clase de rectángulo de muestra contiene tres funciones miembro: *AcceptDetails* , *GetArea* y *Display* .

Instanciar una clase

En el programa anterior, la clase *ExecuteRectangle* contiene el método *Main ()* e instancia el *Rectángulo* clase .

Identificadores

Un identificador es un nombre utilizado para identificar una clase, variable, función o cualquier otro elemento definido por el usuario. Las reglas básicas para nombrar clases en C # son las siguientes:

- Un nombre debe comenzar con una letra que pueda ser seguida por una secuencia de letras, dígitos (0 - 9) o guión bajo. El primer carácter en un identificador no puede ser un dígito.
- No debe contener ningún espacio o símbolo incrustado como? - +! @ #% ^ & * () [] {}. ; : "'/ y \. Sin embargo, se puede utilizar un guión bajo (_).
- No debería ser una palabra clave de C #.

C # Palabras clave

Las palabras clave son palabras reservadas predefinidas para el compilador de C #. Estas palabras clave no se pueden usar como identificadores. Sin embargo, si desea utilizar estas palabras clave como identificadores, puede prefijar la palabra clave con el carácter @.

En C #, algunos identificadores tienen un significado especial en el contexto del código, como get y set se denominan palabras clave contextuales.

La siguiente tabla enumera las palabras clave reservadas y las palabras clave contextuales en C # -

Reserved Keywords						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void

volatile	while					
Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
partial (method)	remove	select	set			

C # - Tipos de datos

Las variables en C # se clasifican en los siguientes tipos:

- Tipos de valor
- Tipos de referencia
- Tipos de puntero

Tipo de valor

A las variables de tipo de valor se les puede asignar un valor directamente. Se derivan de la clase **System.ValueType** .

Los tipos de valor contienen directamente datos. Algunos ejemplos son **int, char y float** , que almacena números, alfabetos y números de coma flotante, respectivamente. Cuando declara un tipo **int** , el sistema asigna memoria para almacenar el valor.

La siguiente tabla enumera los tipos de valores disponibles en C # 2010:

Tipo	Representa	Rango	Valor por defecto

bool	Valor booleano	Verdadero o falso	Falso
byte	Entero sin signo de 8 bits	0 a 255	0 0
carbonizarse	Carácter Unicode de 16 bits	U +0000 a U + ffff	'\ 0'
decimal	Valores decimales precisos de 128 bits con 28-29 dígitos significativos	(-7.9 x 10^{28} a 7.9 x 10^{28}) / 10^{0} a 28	0.0M
doble	Tipo de coma flotante de doble precisión de 64 bits	(+/-) 5.0 x 10^{-324} a (+/-) 1.7 x ^{10308}	0.0D
flotador	Tipo de coma flotante de precisión simple de 32 bits	-3,4 x 10^{38} a + 3,4 x 10^{38}	0.0F
En t	Tipo entero con signo de 32 bits	-2,147,483,648 a 2,147,483,647	0 0
largo	Tipo entero con signo de 64 bits	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	0L
sbyte	Tipo entero con signo de 8 bits	-128 a 127	0 0
corto	Tipo entero con signo de 16 bits	-32,768 a 32,767	0 0
Uint	Tipo entero sin signo de 32 bits	0 a 4,294,967,295	0 0
ulong	Tipo entero sin signo de 64 bits	0 a 18,446,744,073,709,551,615	0 0
ushort	Tipo entero sin signo de 16 bits	0 a 65,535	0 0

Para obtener el tamaño exacto de un tipo o una variable en una plataforma en particular, puede usar el método **sizeof** . La expresión *sizeof (type)* produce el

tamaño de almacenamiento del objeto o escribe bytes. El siguiente es un ejemplo para obtener el tamaño del tipo *int* en cualquier máquina:

```
using System;

namespace DataTypeApplication {

   class Program {

      static void Main(string[] args) {
         Console.WriteLine("Size of int: {0}", sizeof(int));
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Size of int: 4
```

Tipo de referencia

Los tipos de referencia no contienen los datos reales almacenados en una variable, pero contienen una referencia a las variables.

En otras palabras, se refieren a una ubicación de memoria. Usando múltiples variables, los tipos de referencia pueden referirse a una ubicación de memoria. Si una de las variables modifica los datos en la ubicación de la memoria, la otra variable refleja automáticamente este cambio de valor. Ejemplos de tipos de referencia **integrados** son: **objeto** , **dinámico** y **cadena** .

Tipo de objeto

El **tipo de objeto** es la clase base definitiva para todos los tipos de datos en C # Common Type System (CTS). Object es un alias para la clase System.Object. A los tipos de objeto se les pueden asignar valores de cualquier otro tipo, tipos de valor, tipos de referencia, tipos predefinidos o definidos por el usuario. Sin embargo, antes de asignar valores, necesita conversión de tipo.

Cuando un tipo de valor se convierte en un tipo de objeto, se llama **boxeo** y, por otro lado, cuando un tipo de objeto se convierte en un tipo de valor, se llama **unboxing** .

```
object obj;
obj = 100; // this is boxing
```

Tipo dinámico

Puede almacenar cualquier tipo de valor en la variable de tipo de datos dinámico. La verificación de tipos para estos tipos de variables tiene lugar en tiempo de ejecución.

La sintaxis para declarar un tipo dinámico es:

```
dynamic <variable_name> = value;
```

Por ejemplo,

```
dynamic d = 20;
```

Los tipos dinámicos son similares a los tipos de objeto, excepto que la verificación de tipo para las variables de tipo de objeto se realiza en tiempo de compilación, mientras que la de las variables de tipo dinámico se realiza en tiempo de ejecución.

Tipo de cadena

El **tipo de cadena le** permite asignar cualquier valor de cadena a una variable. El tipo de cadena es un alias para la clase System.String. Se deriva del tipo de objeto. El valor para un tipo de cadena se puede asignar usando literales de cadena en dos formas: entre comillas y @ entre comillas.

Por ejemplo,

```
String str = "Postparaprogramadores";
```

Un literal de cadena @quoted tiene el siguiente aspecto:

```
@"Postparaprogramadores";
```

Los tipos de referencia definidos por el usuario son: clase, interfaz o delegado. Discutiremos estos tipos en un capítulo posterior.

Tipo de puntero

Las variables de tipo puntero almacenan la dirección de memoria de otro tipo. Los punteros en C # tienen las mismas capacidades que los punteros en C o C ++.

La sintaxis para declarar un tipo de puntero es:

```
type* identifier;
```

Por ejemplo,

```
char* cptr;
int* iptr;
```

Discutiremos los tipos de puntero en el capítulo 'Códigos inseguros'.

C # - Conversión de tipo

La conversión de tipos es convertir un tipo de datos a otro tipo. También se conoce como Type Casting. En C #, la conversión de tipos tiene dos formas:

- **Conversión de tipo implícita** : estas conversiones las realiza C # de forma segura. Por ejemplo, son conversiones de tipos integrales más pequeños a más grandes y conversiones de clases derivadas a clases base.
- **Conversión de tipo explícita** : estas conversiones se realizan explícitamente por los usuarios que utilizan las funciones predefinidas. Las conversiones explícitas requieren un operador de conversión.

El siguiente ejemplo muestra una conversión de tipo explícito:

```
using System;

namespace TypeConversionApplication {

   class ExplicitConversion {

      static void Main(string[] args) {
         double d = 5673.74;
         int i;

         // cast double to int.
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
5673
```

Métodos de conversión de tipo C

C # proporciona los siguientes métodos de conversión de tipo integrados:

No Señor.	Métodos y descripción
1	**ToBoolean** Convierte un tipo en un valor booleano, donde sea posible.
2	**ToByte** Convierte un tipo en un byte.
3	**ToChar** Convierte un tipo en un único carácter Unicode, donde sea posible.
4 4	**ToDateTime**

	Convierte un tipo (entero o tipo de cadena) en estructuras de fecha y hora.
5 5	**ToDecimal** Convierte un tipo de coma flotante o entero en un tipo decimal.
6 6	**Para duplicar** Convierte un tipo en un tipo doble.
7 7	**ToInt16** Convierte un tipo en un entero de 16 bits.
8	**ToInt32** Convierte un tipo en un entero de 32 bits.
9 9	**ToInt64** Convierte un tipo en un entero de 64 bits.
10	**ToSbyte** Convierte un tipo en un tipo de byte firmado.
11	**ToSingle** Convierte un tipo en un pequeño número de coma flotante.
12	**Encadenar** Convierte un tipo en una cadena.
13	**Digitar** Convierte un tipo en un tipo especificado.
14	**TOUInt16** Convierte un tipo en un tipo int sin signo.
15	**ToUInt32** Convierte un tipo en un tipo largo sin signo.

dieciséis	**ToUInt64** Convierte un tipo en un entero grande sin signo.

El siguiente ejemplo convierte varios tipos de valores en tipo de cadena:

```
using System;

namespace TypeConversionApplication {

   class StringConversion {

      static void Main(string[] args) {
         int i = 75;
         float f = 53.005f;
         double d = 2345.7652;
         bool b = true;

         Console.WriteLine(i.ToString());
         Console.WriteLine(f.ToString());
         Console.WriteLine(d.ToString());
         Console.WriteLine(b.ToString());
         Console.ReadKey();

      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
75
53.005
2345.7652
True
```

C # - Variables

Una variable no es más que un nombre dado a un área de almacenamiento que nuestros programas pueden manipular. Cada variable en C # tiene un tipo específico, que determina el tamaño y el diseño de la memoria de la variable, el rango de valores que se pueden almacenar dentro de esa memoria y el conjunto de operaciones que se pueden aplicar a la variable.

Los tipos de valores básicos proporcionados en C # se pueden clasificar como:

Tipo	Ejemplo
Tipos integrales	sbyte, byte, corto, ushort, int, uint, long, ulong y char

Tipos de punto flotante	flotador y doble
Tipos decimales	decimal
Tipos booleanos	valores verdaderos o falsos, según lo asignado
Tipos anulables	Tipos de datos anulables

C # también permite definir otros tipos de variables de valor como **enum** y tipos de variables de referencia como **class** , que cubriremos en capítulos posteriores.

Definiendo Variables

La sintaxis para la definición de variable en C # es -

```
<data_type> <variable_list>;
```

Aquí, data_type debe ser un tipo de datos válido de C # que incluya char, int, float, double o cualquier tipo de datos definido por el usuario, y variable_list puede consistir en uno o más nombres de identificadores separados por comas.

Aquí se muestran algunas definiciones de variables válidas:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

Puede inicializar una variable en el momento de la definición como -

```
int i = 100;
```

Inicializando Variables

Las variables se inicializan (se les asigna un valor) con un signo igual seguido de una expresión constante. La forma general de inicialización es -

```
variable_name = value;
```

Las variables se pueden inicializar en su declaración. El inicializador consiste en un signo igual seguido de una expresión constante como -

```
<data_type> <variable_name> = value;
```

Algunos ejemplos son:

```
int d = 3, f = 5;     /* initializing d and f. */
```

```
byte z = 22;          /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x';         /* the variable x has the value 'x'. */
```

Es una buena práctica de programación inicializar las variables correctamente, de lo contrario, a veces el programa puede producir resultados inesperados.

El siguiente ejemplo utiliza varios tipos de variables:

```
using System;

namespace VariableDefinition {

   class Program {

      static void Main(string[] args) {
         short a;
         int b ;
         double c;

         /* actual initialization */
         a = 10;
         b = 20;
         c = a + b;
         Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b,
c);
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
a = 10, b = 20, c = 30
```

Aceptar valores del usuario

La clase de **consola** en el espacio de nombres del **sistema** proporciona una función **ReadLine ()** para aceptar la entrada del usuario y almacenarla en una variable.

Por ejemplo,

```
int num;
num = Convert.ToInt32(Console.ReadLine());
```

La función **Convert.ToInt32 ()** convierte los datos ingresados por el usuario al tipo de datos int, porque **Console.ReadLine ()** acepta los datos en formato de cadena.

Expresiones Lvalue y Rvalue en C

Hay dos tipos de expresiones en C #:

- **lvalue** : una expresión que es un lvalue puede aparecer como el lado izquierdo o derecho de una tarea.
- **rvalue** : una expresión que es un rvalue puede aparecer en el lado derecho pero no en el lado izquierdo de una tarea.

Las variables son valores y, por lo tanto, pueden aparecer en el lado izquierdo de una tarea. Los literales numéricos son valores y, por lo tanto, pueden no asignarse y no pueden aparecer en el lado izquierdo. Lo siguiente es una declaración válida de C #:

```
int g = 20;
```

Pero seguir no es una declaración válida y generaría un error en tiempo de compilación:

```
10 = 20;
```

C # - Constantes y literales

Las constantes se refieren a valores fijos que el programa no puede alterar durante su ejecución. Estos valores fijos también se llaman literales. Las constantes pueden ser de cualquiera de los tipos de datos básicos, como una constante entera, una constante flotante, una constante de caracteres o un literal de cadena. También hay constantes de enumeración también.

Las constantes se tratan como variables regulares, excepto que sus valores no pueden modificarse después de su definición.

Literales enteros

Un literal entero puede ser una constante decimal o hexadecimal. Un prefijo especifica la base o la raíz: 0x o 0X para hexadecimal, y no hay identificación de prefijo para decimal.

Un literal entero también puede tener un sufijo que es una combinación de U y L, para unsigned y long, respectivamente. El sufijo puede ser mayúscula o minúscula y puede estar en cualquier orden.

Aquí hay algunos ejemplos de literales enteros:

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
```

Los siguientes son otros ejemplos de varios tipos de literales enteros:

```
85         /* decimal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

Literales de punto flotante

Un literal de coma flotante tiene una parte entera, un punto decimal, una parte fraccionaria y una parte exponente. Puede representar literales de coma flotante en forma decimal o exponencial.

Aquí hay algunos ejemplos de literales de coma flotante:

```
3.14159       /* Legal */
314159E-5F    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Al representar en forma decimal, debe incluir el punto decimal, el exponente o ambos; y mientras se representa usando una forma exponencial, debe incluir la parte entera, la parte fraccional o ambas. El exponente con signo es introducido por e o E.

Constantes de personaje

Los literales de caracteres están encerrados entre comillas simples. Por ejemplo, 'x' y puede almacenarse en una variable simple de tipo char. Un carácter literal puede ser un carácter simple (como 'x'), una secuencia de escape (como '\ t') o un carácter universal (como '\ u02C0').

Hay ciertos caracteres en C # cuando están precedidos por una barra invertida. Tienen un significado especial y se usan para representar como nueva línea (\ n) o tabulación (\ t). Aquí hay una lista de algunos de estos códigos de secuencia de escape:

Secuencia de escape	Sentido
\\	\ personaje
\ '	' personaje
\ "	" personaje
\?	? personaje
\un	Alerta o campana
\si	Retroceso
\F	Alimentación de formulario

\norte	Nueva línea
\ r	Retorno de carro
\ t	Pestaña horizontal
\ v	Pestaña vertical
\ xhh. . .	Número hexadecimal de uno o más dígitos.

El siguiente es el ejemplo para mostrar algunos caracteres de secuencia de escape:

```
using System;

namespace EscapeChar {

   class Program {

      static void Main(string[] args) {
         Console.WriteLine("Hello\tWorld\n\n");
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Hello   World
```

Literales de cuerda

Los literales de cadena o las constantes están entre comillas dobles "" o con @ "". Una cadena contiene caracteres que son similares a los literales de caracteres: caracteres simples, secuencias de escape y caracteres universales.

Puede dividir una línea larga en varias líneas usando literales de cadena y separando las partes usando espacios en blanco.

Aquí hay algunos ejemplos de literales de cadena. Las tres formas son cadenas idénticas.

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
@"hello dear"
```

Definiendo constantes

Las constantes se definen con la palabra clave **const** . La sintaxis para definir una constante es:

```
const <data_type> <constant_name> = value;
```

El siguiente programa muestra cómo definir y usar una constante en su programa:

```
using System;

namespace DeclaringConstants {

    class Program {

        static void Main(string[] args) {
            const double pi = 3.14159;

            // constant declaration
            double r;
            Console.WriteLine("Enter Radius: ");
            r = Convert.ToDouble(Console.ReadLine());
            double areaCircle = pi * r * r;
            Console.WriteLine("Radius: {0}, Area: {1}", r,
areaCircle);
            Console.ReadLine();
        }
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Enter Radius:
3
Radius: 3, Area: 28.27431
```

C # - Operadores

Un operador es un símbolo que le dice al compilador que realice manipulaciones matemáticas o lógicas específicas. C # tiene un amplio conjunto de operadores integrados y proporciona el siguiente tipo de operadores:

- Operadores aritméticos
- Operadores relacionales
- Operadores logicos
- Operadores bit a bit
- Operadores de Asignación
- Operadores diversos

Este tutorial explica los operadores aritméticos, relacionales, lógicos, bit a bit, de asignación y otros operadores uno por uno.

Operadores aritméticos

La siguiente tabla muestra todos los operadores aritméticos admitidos por C #. Suponga que la variable **A** contiene 10 y la variable **B** contiene 20 y luego -

Mostrar ejemplos

Operador	Descripción	Ejemplo
+	Agrega dos operandos	A + B = 30
-	Resta el segundo operando del primero	A - B = -10
* *	Multiplica ambos operandos	A * B = 200
/ /	Divide el numerador entre el numerador	B / A = 2
%	Operador de módulo y resto después de una división entera	B% A = 0
++	El operador de incremento aumenta el valor entero en uno	A ++ = 11
-	El operador de disminución disminuye el valor entero en uno	A-- = 9

Operadores relacionales

La siguiente tabla muestra todos los operadores relacionales compatibles con C #. Suponga que la variable **A** tiene 10 y la variable **B** tiene 20, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
==	Comprueba si los valores de dos operandos son iguales o no, en caso afirmativo, la condición se vuelve verdadera.	(A == B) no es cierto.
! =	Comprueba si los valores de dos operandos son iguales o no, si los valores no son iguales, la condición se vuelve verdadera.	(A! = B) es cierto.

>	Comprueba si el valor del operando izquierdo es mayor que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A> B) no es cierto.
<	Comprueba si el valor del operando izquierdo es menor que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A <B) es cierto.
> =	Comprueba si el valor del operando izquierdo es mayor o igual que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A> = B) no es cierto.
<=	Comprueba si el valor del operando izquierdo es menor o igual que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A <= B) es cierto.

Operadores logicos

La siguiente tabla muestra todos los operadores lógicos admitidos por C #. Suponga que la variable **A** tiene un valor booleano verdadero y la variable **B** tiene un valor booleano falso, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
&&	Llamado operador lógico AND. Si ambos operandos son distintos de cero, la condición se vuelve verdadera.	(A y B) es falso.
||	Llamado Lógico O Operador. Si alguno de los dos operandos es distinto de cero, la condición se vuelve verdadera.	(A || B) es cierto.
!	Llamado operador lógico NO. Se usa para invertir el estado lógico de su operando. Si una condición es verdadera, entonces el operador lógico NO hará falso.	! (A y B) es cierto.

Operadores bit a bit

El operador bit a bit trabaja en bits y realiza operaciones bit a bit. Las tablas de verdad para &, | y ^ son las siguientes:

pag	q	p & q	p | q	p ^ q
0 0	0 0	0 0	0 0	0 0
0 0	1	0 0	1	1
1	1	1	1	0 0
1	0 0	0 0	1	1

Suponga que si A = 60; y B = 13; entonces en formato binario son los siguientes:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A | B = 0011 1101

A ^ B = 0011 0001

~ A = 1100 0011

Los operadores Bitwise admitidos por C # se enumeran en la siguiente tabla. Suponga que la variable A tiene 60 y la variable B tiene 13, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
Y	El operador binario AND copia un poco al resultado si existe en ambos operandos.	(A y B) = 12, que es 0000 1100
El |	El operador binario O copia un bit si existe en cualquiera de los operandos.	(A | B) = 61, que es 0011 1101
^	El operador binario XOR copia el bit si está establecido en un operando pero no en ambos.	(A ^ B) = 49, que es 0011 0001

~	El operador de complemento de binarios es unario y tiene el efecto de "voltear" los bits.	(~ A) = -61, que es 1100 0011 en el complemento de 2 debido a un número binario con signo.
<<	Operador binario de desplazamiento a la izquierda. El valor de los operandos de la izquierda se mueve hacia la izquierda por la cantidad de bits especificados por el operando de la derecha.	A << 2 = 240, que es 1111 0000
>>	Operador binario de desplazamiento a la derecha. El valor de los operandos de la izquierda se mueve hacia la derecha por la cantidad de bits especificados por el operando de la derecha.	A >> 2 = 15, que es 0000 1111

Operadores de Asignación

Los siguientes operadores de asignación son compatibles con C #:

Mostrar ejemplos

Operador	Descripción	Ejemplo
=	Operador de asignación simple, asigna valores de operandos del lado derecho al operando del lado izquierdo	C = A + B asigna el valor de A + B a C
+ =	Agregar operador de asignación AND, agrega el operando derecho al operando izquierdo y asigna el resultado al operando izquierdo	C + = A es equivalente a C = C + A
- =	Restar operador de asignación AND, resta el operando derecho del operando izquierdo y asigna el resultado al operando izquierdo	C - = A es equivalente a C = C - A
* =	Operador de multiplicación Y asignación, multiplica el operando derecho con el operando izquierdo y asigna el resultado al operando izquierdo	C * = A es equivalente a C = C * A

/ =	Operador de división Y asignación, divide el operando izquierdo con el operando derecho y asigna el resultado al operando izquierdo	C / = A es equivalente a C = C / A
% =	Operador de asignación de módulo Y, toma módulo usando dos operandos y asigna el resultado al operando izquierdo	C% = A es equivalente a C = C% A
<< =	Desplazamiento a la izquierda y operador de asignación	C << = 2 es lo mismo que C = C << 2
>> =	Desplazamiento a la derecha y operador de asignación	C >> = 2 es lo mismo que C = C >> 2
& =	Operador de asignación Y bit a bit	C & = 2 es lo mismo que C = C & 2
^ =	OR exclusivo bit a bit y operador de asignación	C ^ = 2 es lo mismo que C = C ^ 2
| =	OR inclusivo a nivel de bit y operador de asignación	C | = 2 es lo mismo que C = C | 2

Operadores Misceláneos

Hay algunos otros operadores importantes que incluyen **sizeof, typeof** y **? :** apoyado por C #.

Mostrar ejemplos

Operador	Descripción	Ejemplo

tamaño de()	Devuelve el tamaño de un tipo de datos.	sizeof (int), devuelve 4.
tipo de()	Devuelve el tipo de una clase.	typeof (StreamReader);
Y	Devuelve la dirección de una variable.	&un; Devuelve la dirección real de la variable.
* *	Puntero a una variable.	*un; crea un puntero llamado 'a' a una variable.
? :	Expresión condicional	Si la condición es verdadera? Entonces valor X: de lo contrario, valor Y
es	Determina si un objeto es de cierto tipo.	If (Ford is Car) // comprueba si Ford es un objeto de la clase Car.
como	Lanza sin provocar una excepción si falla el lanzamiento.	Object obj = new StringReader ("Hola"); StringReader r = obj como StringReader;

Precedencia del operador en C

La precedencia del operador determina la agrupación de términos en una expresión. Esto afecta la evaluación de una expresión. Ciertos operadores tienen mayor prioridad que otros; por ejemplo, el operador de multiplicación tiene mayor prioridad que el operador de suma.

Por ejemplo x = 7 + 3 * 2; aquí, a x se le asigna 13, no 20 porque el operador * tiene mayor prioridad que +, por lo que la primera evaluación se realiza para 3 * 2 y luego se agrega 7.

Aquí, los operadores con la precedencia más alta aparecen en la parte superior de la tabla, aquellos con la más baja aparecen en la parte inferior. Dentro de una expresión, los operadores de mayor precedencia se evalúan primero.

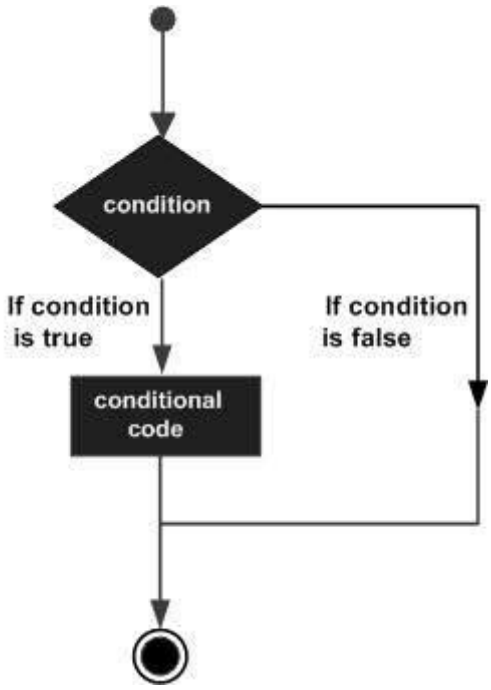
Mostrar ejemplos

Categoría	Operador	Asociatividad
Sufijo	() [] ->. ++ - -	De izquierda a derecha
Unario	+ -! ~ ++ - - (tipo) * y sizeof	De derecha a izquierda
Multiplicativo	* /%	De izquierda a derecha
Aditivo	+ -	De izquierda a derecha
Cambio	<< >>	De izquierda a derecha
Relacional	<<=>> =	De izquierda a derecha
Igualdad	==! =	De izquierda a derecha
Bitwise Y	Y	De izquierda a derecha
Bitwise XOR	^	De izquierda a derecha
Bitwise O	El |	De izquierda a derecha
Y lógico	&&	De izquierda a derecha
O lógico	||	De izquierda a derecha
Condicional	?:	De derecha a izquierda
Asignación	= + = - = * = / =% = >> = << = & = ^ = | =	De derecha a izquierda
Coma	,	De izquierda a derecha

C # - Toma de decisiones

Las estructuras de toma de decisiones requieren que el programador especifique una o más condiciones para ser evaluadas o probadas por el programa, junto con una declaración o declaraciones que se ejecutarán si se determina que la condición es verdadera, y opcionalmente, otras declaraciones que se ejecutarán si la condición se determina que es falso

A continuación se presenta la forma general de una estructura de toma de decisiones típica que se encuentra en la mayoría de los lenguajes de programación:



C # proporciona los siguientes tipos de declaraciones de toma de decisiones. Haga clic en los siguientes enlaces para verificar sus detalles.

No Señor.	Declaración y descripción
1	si la declaración Una **declaración if** consiste en una expresión booleana seguida de una o más declaraciones.
2	si ... otra declaración Una **instrucción if** puede ser seguida por una **instrucción else** opcional , que se ejecuta cuando la expresión booleana es falsa.
3	instrucciones if anidadas Puede usar una declaración **if** o **else if** dentro de otra declaración **if** o **else if** (s).
4 4	declaración de cambio

	Una declaración de **cambio** permite que una variable sea probada para la igualdad contra una lista de valores.
5 5	instrucciones de cambio anidadas Puede utilizar uno **interruptor de** declaración dentro de otro **interruptor de** declaración (s).

Los ? : Operador

Hemos cubierto **operador condicional? :** en el capítulo anterior que se puede usar para reemplazar las declaraciones **if ... else** . Tiene la siguiente forma general:

```
Exp1 ? Exp2 : Exp3;
```

Donde Exp1, Exp2 y Exp3 son expresiones. Observe el uso y la colocación del colon.

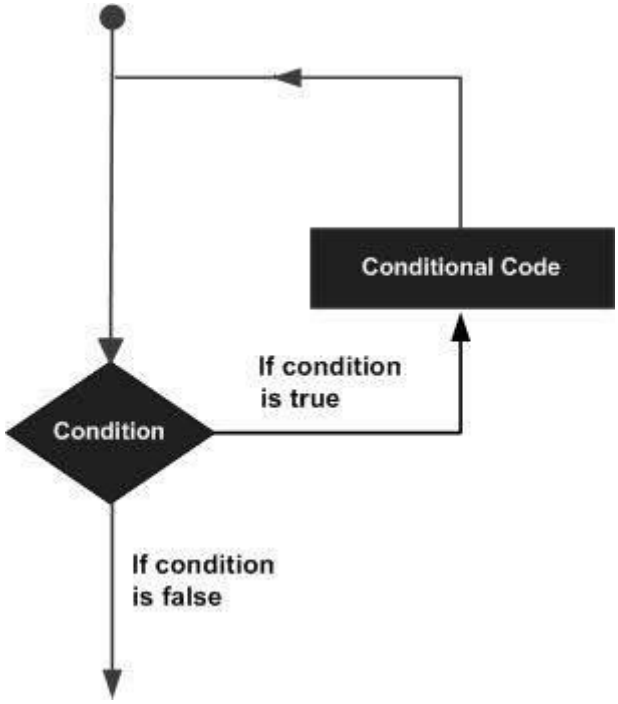
El valor de a? la expresión se determina como sigue: se evalúa Exp1. Si es cierto, ¿se evalúa Exp2 y se convierte en el valor de todo? expresión. Si Exp1 es falso, se evalúa Exp3 y su valor se convierte en el valor de la expresión.

C # - Bucles

Puede haber una situación en la que necesite ejecutar un bloque de código varias veces. En general, las instrucciones se ejecutan secuencialmente: la primera instrucción de una función se ejecuta primero, seguida de la segunda, y así sucesivamente.

Los lenguajes de programación proporcionan diversas estructuras de control que permiten rutas de ejecución más complicadas.

Una declaración de bucle nos permite ejecutar una declaración o un grupo de declaraciones varias veces y lo siguiente es lo general de una declaración de bucle en la mayoría de los lenguajes de programación:



C # proporciona los siguientes tipos de bucle para manejar los requisitos de bucle. Haga clic en los siguientes enlaces para verificar sus detalles.

No Señor.	Tipo de bucle y descripción
1	mientras bucle Repite una declaración o un grupo de declaraciones mientras una condición dada es verdadera. Prueba la condición antes de ejecutar el cuerpo del bucle.
2	en bucle Ejecuta una secuencia de declaraciones varias veces y abrevia el código que administra la variable de bucle.
3	hacer ... mientras bucle Es similar a una instrucción while, excepto que prueba la condición al final del cuerpo del bucle
4 4	bucles anidados Puede usar uno o más bucles dentro de otro mientras, para o hacer ... while loop.

Declaraciones de control de bucle

Las instrucciones de control de bucle cambian la ejecución de su secuencia normal. Cuando la ejecución deja un ámbito, todos los objetos automáticos que se crearon en ese ámbito se destruyen.

C # proporciona las siguientes declaraciones de control. Haga clic en los siguientes enlaces para verificar sus detalles.

No Señor.	Declaración de control y descripción
1	declaración de ruptura Termina la instrucción **loop** o **switch** y transfiere la ejecución a la instrucción que sigue inmediatamente al loop o switch.
2	continuar declaración Hace que el bucle omita el resto de su cuerpo e inmediatamente vuelva a probar su condición antes de reiterar.

Bucle infinito

Un bucle se convierte en bucle infinito si una condición nunca se vuelve falsa. El bucle **for** se usa tradicionalmente para este propósito. Como no se requiere ninguna de las tres expresiones que forman el bucle for, puede hacer un bucle sin fin dejando vacía la expresión condicional.

Ejemplo

```
using System;

namespace Loops {

   class Program {

      static void Main(string[] args) {
         for (; ; ) {
            Console.WriteLine("Hey! I am Trapped");
         }
      }
   }
}
```

Cuando la expresión condicional está ausente, se supone que es verdadera. Es posible que tenga una expresión de inicialización e incremento, pero los programadores usan más comúnmente la construcción for (;;) para significar un bucle infinito.

C # - Encapsulación

La encapsulación se define "como el proceso de encerrar uno o más elementos dentro de un paquete físico o lógico". La encapsulación, en la metodología de programación orientada a objetos, impide el acceso a los detalles de implementación.

La abstracción y la encapsulación son características relacionadas en la programación orientada a objetos. La abstracción permite hacer visible la información relevante y la encapsulación permite al programador *implementar el nivel deseado de abstracción* .

La encapsulación se implementa mediante el uso de **especificadores de acceso** . Un **especificador de acceso** define el alcance y la visibilidad de un miembro de la clase. C # admite los siguientes especificadores de acceso:

- Público
- Privado
- Protegido
- Interno
- Interno protegido

Especificador de acceso público

El especificador de acceso público permite que una clase exponga sus variables miembro y funciones miembro a otras funciones y objetos. Se puede acceder a cualquier miembro público desde fuera de la clase.

El siguiente ejemplo ilustra esto:

```
using System;

namespace RectangleApplication {

   class Rectangle {
      //member variables
      public double length;
      public double width;

      public double GetArea() {
         return length * width;
      }

      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle

   class ExecuteRectangle {
      static void Main(string[] args) {
         Rectangle r = new Rectangle();
         r.length = 4.5;
         r.width = 3.5;
         r.Display();
         Console.ReadLine();
      }
   }
```

```
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

En el ejemplo anterior, las variables miembro longitud y anchura se declaran **públicas** , por lo que se puede acceder a ellas desde la función Main () utilizando una instancia de la clase Rectangle, denominada **r** .

La función miembro *Display ()* y *GetArea ()* también puede acceder a estas variables directamente sin utilizar ninguna instancia de la clase.

Las funciones miembro *Display ()* también se declaran **públicas** , por lo que también se puede acceder desde *Main ()* utilizando una instancia de la clase Rectangle, llamada **r** .

Especificador de acceso privado

El especificador de acceso privado permite que una clase oculte sus variables miembro y funciones miembro de otras funciones y objetos. Solo las funciones de la misma clase pueden acceder a sus miembros privados. Incluso una instancia de una clase no puede acceder a sus miembros privados.

El siguiente ejemplo ilustra esto:

```
using System;

namespace RectangleApplication {

   class Rectangle {
      //member variables
      private double length;
      private double width;

      public void Acceptdetails() {
         Console.WriteLine("Enter Length: ");
         length = Convert.ToDouble(Console.ReadLine());
         Console.WriteLine("Enter Width: ");
         width = Convert.ToDouble(Console.ReadLine());
      }

      public double GetArea() {
         return length * width;
      }

      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle
```

```
   class ExecuteRectangle {
      static void Main(string[] args) {
         Rectangle r = new Rectangle();
         r.Acceptdetails();
         r.Display();
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Enter Length:
4.4
Enter Width:
3.3
Length: 4.4
Width: 3.3
Area: 14.52
```

En el ejemplo anterior, las variables miembro longitud y anchura se declaran **privadas** , por lo que no se puede acceder desde la función Main (). Las funciones miembro *AcceptDetails ()* y *Display ()* pueden acceder a estas variables. Dado que las funciones miembro *AcceptDetails ()* y *Display ()* se declaran **públicas** , se puede acceder a ellas desde *Main ()* utilizando una instancia de la clase Rectangle, llamada **r** .

Especificador de acceso protegido

El especificador de acceso protegido permite que una clase secundaria acceda a las variables miembro y a las funciones miembro de su clase base. De esta manera, ayuda a implementar la herencia. Discutiremos esto con más detalles en el capítulo de herencia.

Especificador de acceso interno

El especificador de acceso interno permite que una clase exponga sus variables miembro y funciones miembro a otras funciones y objetos en el ensamblaje actual. En otras palabras, se puede acceder a cualquier miembro con especificador de acceso interno desde cualquier clase o método definido dentro de la aplicación en la que se define el miembro.

El siguiente programa ilustra esto:

```
using System;

namespace RectangleApplication {

   class Rectangle {
      //member variables
      internal double length;
      internal double width;
```

```
      double GetArea() {
         return length * width;
      }

      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle

   class ExecuteRectangle {
      static void Main(string[] args) {
         Rectangle r = new Rectangle();
         r.length = 4.5;
         r.width = 3.5;
         r.Display();
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

En el ejemplo anterior, observe que la función miembro *GetArea ()* no se declara con ningún especificador de acceso. Entonces, ¿cuál sería el especificador de acceso predeterminado de un miembro de la clase si no mencionamos ninguno? Es **privado** .

Especificador de acceso interno protegido

El especificador de acceso interno protegido permite que una clase oculte sus variables miembro y funciones miembro de otros objetos y funciones de clase, excepto una clase hija dentro de la misma aplicación. Esto también se usa al implementar la herencia.

C # - Métodos

Un método es un grupo de declaraciones que juntas realizan una tarea. Cada programa C # tiene al menos una clase con un método llamado Main.

Para usar un método, necesita:

- Define el método
- Llamar al método

Definición de métodos en C

Cuando define un método, básicamente declara los elementos de su estructura. La sintaxis para definir un método en C # es la siguiente:

```
<Access Specifier> <Return Type> <Method Name>(Parameter
List) {
   Method Body
}
```

Los siguientes son los diversos elementos de un método:

- **Especificador de acceso** : determina la visibilidad de una variable o un método de otra clase.
- **Tipo de retorno** : un método puede devolver un valor. El tipo de retorno es el tipo de datos del valor que devuelve el método. Si el método no devuelve ningún valor, entonces el tipo de retorno es **nulo** .
- **Nombre del método: el nombre del** método es un identificador único y distingue entre mayúsculas y minúsculas. No puede ser el mismo que cualquier otro identificador declarado en la clase.
- **Lista de parámetros** : encerrados entre paréntesis, los parámetros se utilizan para pasar y recibir datos de un método. La lista de parámetros se refiere al tipo, orden y número de parámetros de un método. Los parámetros son opcionales; es decir, un método puede no contener parámetros.
- **Cuerpo del método** : contiene el conjunto de instrucciones necesarias para completar la actividad requerida.

Ejemplo

El siguiente fragmento de código muestra una función *FindMax* que toma dos valores enteros y devuelve el mayor de los dos. Tiene un especificador de acceso público, por lo que se puede acceder desde fuera de la clase utilizando una instancia de la clase.

```
class NumberManipulator {

   public int FindMax(int num1, int num2) {
      /* local variable declaration */
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
   ...
}
```

Métodos de llamada en C

Puede llamar a un método usando el nombre del método. El siguiente ejemplo ilustra esto:

```
using System;

namespace CalculatorApplication {

   class NumberManipulator {

      public int FindMax(int num1, int num2) {
         /* local variable declaration */
         int result;

         if (num1 > num2)
            result = num1;
         else
            result = num2;
         return result;
      }

      static void Main(string[] args) {
         /* local variable definition */
         int a = 100;
         int b = 200;
         int ret;
         NumberManipulator n = new NumberManipulator();

         //calling the FindMax method
         ret = n.FindMax(a, b);
         Console.WriteLine("Max value is : {0}", ret );
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Max value is : 200
```

También puede llamar al método público desde otras clases utilizando la instancia de la clase. Por ejemplo, el método *FindMax* pertenece a la clase *NumberManipulator* , puede llamarlo desde otra clase *Test* .

```
using System;

namespace CalculatorApplication {

   class NumberManipulator {

      public int FindMax(int num1, int num2) {
         /* local variable declaration */
         int result;

         if(num1 > num2)
            result = num1;
         else
            result = num2;
```

```
            return result;
         }
      }

      class Test {

         static void Main(string[] args) {
            /* local variable definition */
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();

            //calling the FindMax method
            ret = n.FindMax(a, b);
            Console.WriteLine("Max value is : {0}", ret );
            Console.ReadLine();
         }
      }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Max value is : 200
```

Llamada al método recursivo

Un método puede llamarse a sí mismo. Esto se conoce como **recursión** . El siguiente es un ejemplo que calcula el factorial para un número dado usando una función recursiva:

```
using System;

namespace CalculatorApplication {

   class NumberManipulator {

      public int factorial(int num) {
         /* local variable declaration */
         int result;
         if (num == 1) {
            return 1;
         }
         else {
            result = factorial(num - 1) * num;
            return result;
         }
      }

      static void Main(string[] args) {
         NumberManipulator n = new NumberManipulator();
```

```
         //calling the factorial method {0}",
n.factorial(6));
         Console.WriteLine("Factorial of 7 is : {0}",
n.factorial(7));
         Console.WriteLine("Factorial of 8 is : {0}",
n.factorial(8));
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320
```

Pasar parámetros a un método

Cuando se llama al método con parámetros, debe pasar los parámetros al método. Hay tres formas en que los parámetros se pueden pasar a un método:

No Señor.	Mecanismo y Descripción
1	Parámetros de valor Este método copia el valor real de un argumento en el parámetro formal de la función. En este caso, los cambios realizados en el parámetro dentro de la función no tienen efecto en el argumento.
2	Parámetros de referencia Este método copia la referencia a la ubicación de memoria de un argumento en el parámetro formal. Esto significa que los cambios realizados en el parámetro afectan el argumento.
3	Parámetros de salida Este método ayuda a devolver más de un valor.

C # - Nulables

C # proporciona tipos de datos especiales, los tipos **anulables** , a los que puede asignar un rango normal de valores, así como valores nulos.

Por ejemplo, puede almacenar cualquier valor de -2,147,483,648 a 2,147,483,647 o nulo en una variable Nullable <Int32>. Del mismo modo, puede asignar verdadero, falso o nulo en una variable <bool> anulable. La sintaxis para declarar un tipo **anulable** es la siguiente:

```
< data_type> ? <variable_name> = null;
```

El siguiente ejemplo demuestra el uso de tipos de datos anulables:

```
using System;

namespace CalculatorApplication {

   class NullablesAtShow {

      static void Main(string[] args) {
         int? num1 = null;
         int? num2 = 45;
         double? num3 = new double?();
         double? num4 = 3.14157;

         bool? boolval = new bool?();

         // display the values
         Console.WriteLine("Nullables at Show: {0}, {1}, {2},
{3}", num1, num2, num3, num4);
         Console.WriteLine("A Nullable boolean value: {0}",
boolval);
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Nullables at Show: , 45,  , 3.14157
A Nullable boolean value:
```

El operador de fusión nula (??)

El operador de fusión nula se utiliza con los tipos de valores anulables y los tipos de referencia. Se utiliza para convertir un operando al tipo de otro operando de tipo de valor anulable (o no), donde es posible una conversión implícita.

Si el valor del primer operando es nulo, el operador devuelve el valor del segundo operando; de lo contrario, devuelve el valor del primer operando. El siguiente ejemplo explica esto:

```
using System;

namespace CalculatorApplication {

   class NullablesAtShow {

      static void Main(string[] args) {
         double? num1 = null;
         double? num2 = 3.14157;
         double num3;
         num3 = num1 ?? 5.34;
         Console.WriteLine(" Value of num3: {0}", num3);
```

```
            num3 = num2 ?? 5.34;
            Console.WriteLine(" Value of num3: {0}", num3);
            Console.ReadLine();
        }
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

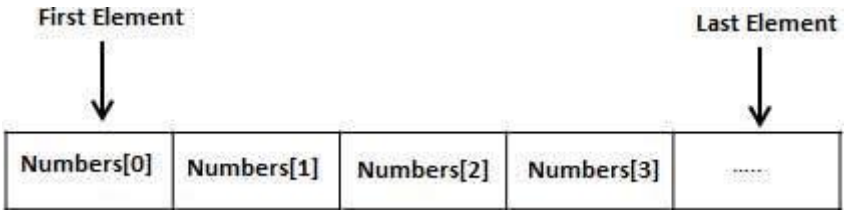
```
Value of num3: 5.34
Value of num3: 3.14157
```

C #: matrices

Una matriz almacena una colección secuencial de tamaño fijo de elementos del mismo tipo. Una matriz se usa para almacenar una colección de datos, pero a menudo es más útil pensar en una matriz como una colección de variables del mismo tipo almacenadas en ubicaciones de memoria contiguas.

En lugar de declarar variables individuales, como número0, número1, ... y número99, declara una variable de matriz como números y usa números [0], números [1] y ..., números [99] para representar variables individuales Se accede a un elemento específico en una matriz mediante un índice.

Todas las matrices consisten en ubicaciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la dirección más alta al último elemento.



Declarar matrices

Para declarar una matriz en C #, puede usar la siguiente sintaxis:

```
datatype[] arrayName;
```

dónde,

- *tipo de datos* se utiliza para especificar el tipo de elementos en la matriz.
- *[]* especifica el rango de la matriz. El rango especifica el tamaño de la matriz.
- *arrayName* especifica el nombre de la matriz.

Por ejemplo,

```
double[] balance;
```

Inicializando una matriz

Declarar una matriz no inicializa la matriz en la memoria. Cuando se inicializa la variable de matriz, puede asignar valores a la matriz.

La matriz es un tipo de referencia, por lo que debe usar la **nueva** palabra clave para crear una instancia de la matriz. Por ejemplo,

```
double[] balance = new double[10];
```

Asignación de valores a una matriz

Puede asignar valores a elementos de matriz individuales, utilizando el número de índice, como -

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

Puede asignar valores a la matriz en el momento de la declaración, como se muestra:

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

También puede crear e inicializar una matriz, como se muestra:

```
int [] marks = new int[5]  { 99,  98, 92, 97, 95};
```

También puede omitir el tamaño de la matriz, como se muestra:

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
```

Puede copiar una variable de matriz en otra variable de matriz de destino. En tal caso, tanto el destino como el origen apuntan a la misma ubicación de memoria:

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
int[] score = marks;
```

Cuando crea una matriz, el compilador de C # inicializa implícitamente cada elemento de la matriz a un valor predeterminado según el tipo de matriz. Por ejemplo, para una matriz int todos los elementos se inicializan a 0.

Acceso a elementos de matriz

Se accede a un elemento indexando el nombre de la matriz. Esto se hace colocando el índice del elemento entre corchetes después del nombre de la matriz. Por ejemplo,

```
double salary = balance[9];
```

El siguiente ejemplo muestra los conceptos mencionados anteriormente, la declaración, la asignación y el acceso a las matrices:

```
using System;

namespace ArrayApplication {

   class MyArray {
```

```
      static void Main(string[] args) {
         int []  n = new int[10]; /* n is an array of 10
integers */
         int i,j;

         /* initialize elements of array n */
         for ( i = 0; i < 10; i++ ) {
            n[ i ] = i + 100;
         }

         /* output each array element's value */
         for (j = 0; j < 10; j++ ) {
            Console.WriteLine("Element[{0}] = {1}", j, n[j]);
         }
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Usando el bucle *foreach*

En el ejemplo anterior, utilizamos un bucle for para acceder a cada elemento de la matriz. También puede usar una instrucción **foreach** para recorrer en iteración una matriz.

```
using System;

namespace ArrayApplication {

   class MyArray {

      static void Main(string[] args) {
         int []  n = new int[10]; /* n is an array of 10
integers */

         /* initialize elements of array n */
         for ( int i = 0; i < 10; i++ ) {
            n[i] = i + 100;
         }
```

```
            /* output each array element's value */
            foreach (int j in n ) {
               int i = j-100;
               Console.WriteLine("Element[{0}] = {1}", i, j);

            }
            Console.ReadKey();
         }
      }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Matrices C

A continuación se detallan algunos conceptos importantes relacionados con la matriz que deben ser claros para un programador de C #:

No Señor.	Concepto y descripción
1	Matrices multidimensionales C # admite matrices multidimensionales. La forma más simple de la matriz multidimensional es la matriz bidimensional.
2	Matrices dentadas C # admite matrices multidimensionales, que son matrices de matrices.
3	Pasar matrices a funciones Puede pasar a la función un puntero a una matriz especificando el nombre de la matriz sin un índice.
4 4	Matrices de parámetros Esto se utiliza para pasar un número desconocido de parámetros a una función.
5 5	La clase de matriz

	Definido en el espacio de nombres del sistema, es la clase base para todas las matrices y proporciona varias propiedades y métodos para trabajar con matrices.

C # - Cuerdas

En C #, puede usar cadenas como matriz de caracteres. Sin embargo, la práctica más común es usar la palabra clave de **cadena** para declarar una variable de cadena. La palabra clave string es un alias para la clase **System.String** .

Crear un objeto de cadena

Puede crear un objeto de cadena utilizando uno de los siguientes métodos:

- Al asignar un literal de cadena a una variable de cadena
- Mediante el uso de un constructor de clase String
- Mediante el uso del operador de concatenación de cadenas (+)
- Al recuperar una propiedad o llamar a un método que devuelve una cadena
- Al llamar a un método de formato para convertir un valor o un objeto a su representación de cadena

El siguiente ejemplo demuestra esto:

```
using System;

namespace StringApplication {

   class Program {

      static void Main(string[] args) {
         //from string literal and string concatenation
         string fname, lname;
         fname = "Rowan";
         lname = "Atkinson";

         char []letters= { 'H', 'e', 'l', 'l','o' };
         string [] sarray={ "Hello", "From", "Tutorials",
"Point" };

         string fullname = fname + lname;
         Console.WriteLine("Full Name: {0}", fullname);

         //by using string constructor { 'H', 'e', 'l',
'l','o' };
         string greetings = new string(letters);
         Console.WriteLine("Greetings: {0}", greetings);

         //methods returning string { "Hello", "From",
"Tutorials", "Point" };
         string message = String.Join(" ", sarray);
```

```
         Console.WriteLine("Message: {0}", message);

         //formatting method to convert a value
         DateTime waiting = new DateTime(2012, 10, 10, 17,
58, 1);
         string chat = String.Format("Message sent at {0:t}
on {0:D}", waiting);
         Console.WriteLine("Message: {0}", chat);
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Full Name: RowanAtkinson
Greetings: Hello
Message: Hello From Postparaprogramadores
Message: Message sent at 5:58 PM on Wednesday, October 10,
2012
```

Propiedades de la clase de cadena

La clase String tiene las siguientes dos propiedades:

No Señor.	Descripción de propiedad
1	**Caracteres** Obtiene el objeto *Char* en una posición especificada en el objeto *String* actual .
2	**Longitud** Obtiene el número de caracteres en el objeto de cadena actual.

Métodos de la clase de cadena

La clase String tiene numerosos métodos que lo ayudan a trabajar con los objetos de cadena. La siguiente tabla proporciona algunos de los métodos más utilizados:

A continuación se muestra la lista de métodos de la clase String.

No Señor.	Métodos y descripción
1	**public static int Compare (cadena strA, cadena strB)** Compara dos objetos de cadena especificados y devuelve un número entero

	que indica su posición relativa en el orden de clasificación.
2	**public static int Compare (cadena strA, cadena strB, bool ignoreCase)** Compara dos objetos de cadena especificados y devuelve un número entero que indica su posición relativa en el orden de clasificación. Sin embargo, ignora el caso si el parámetro booleano es verdadero.
3	**cadena estática pública Concat (cadena str0, cadena str1)** Concatena dos objetos de cadena.
4 4	**cadena estática pública Concat (cadena str0, cadena str1, cadena str2)** Concatena tres objetos de cadena.
5 5	**cadena estática pública Concat (cadena str0, cadena str1, cadena str2, cadena str3)** Concatena cuatro objetos de cadena.
6 6	**public bool Contins (valor de cadena)** Devuelve un valor que indica si el objeto String especificado se produce dentro de esta cadena.
7 7	**Copia de cadena estática pública (cadena de cadena)** Crea un nuevo objeto String con el mismo valor que la cadena especificada.
8	**public void CopyTo (int sourceIndex, char [] destination, int destinationIndex, int count)** Copia un número específico de caracteres desde una posición específica del objeto String a una posición específica en una matriz de caracteres Unicode.
9 9	**public bool EndsWith (valor de cadena)** Determina si el final del objeto de cadena coincide con la cadena especificada.
10	**public bool Equals (valor de cadena)** Determina si el objeto String actual y el objeto String especificado tienen el mismo valor.

11	**public static bool Equals (cadena a, cadena b)** Determina si dos objetos String especificados tienen el mismo valor.
12	**Formato de cadena estática pública (formato de cadena, Objeto arg0)** Reemplaza uno o más elementos de formato en una cadena especificada con la representación de cadena de un objeto especificado.
13	**public int IndexOf (valor de char)** Devuelve el índice de base cero de la primera aparición del carácter Unicode especificado en la cadena actual.
14	**public int IndexOf (valor de cadena)** Devuelve el índice de base cero de la primera aparición de la cadena especificada en esta instancia.
15	**public int IndexOf (valor de char, int startIndex)** Devuelve el índice basado en cero de la primera aparición del carácter Unicode especificado en esta cadena, comenzando la búsqueda en la posición del carácter especificado.
dieciséis	**public int IndexOf (valor de cadena, int startIndex)** Devuelve el índice basado en cero de la primera aparición de la cadena especificada en esta instancia, comenzando la búsqueda en la posición de carácter especificada.
17	**public int IndexOfAny (char [] anyOf)** Devuelve el índice basado en cero de la primera aparición en esta instancia de cualquier carácter en una matriz especificada de caracteres Unicode.
18 años	**public int IndexOfAny (char [] anyOf, int startIndex)** Devuelve el índice basado en cero de la primera aparición en esta instancia de cualquier carácter en una matriz especificada de caracteres Unicode, comenzando la búsqueda en la posición de carácter especificada.
19	**Insertar cadena pública (int startIndex, valor de cadena)** Devuelve una nueva cadena en la que se inserta una cadena especificada en una posición de índice especificada en el objeto de cadena actual.

20	**public static bool IsNullOrEmpty (valor de cadena)** Indica si la cadena especificada es nula o una cadena vacía.
21	**cadena estática pública Join (separador de cadena, valor de cadena de parámetros [])** Concatena todos los elementos de una matriz de cadenas, utilizando el separador especificado entre cada elemento.
22	**unión de cadena estática pública (separador de cadena, valor de cadena [], int startIndex, int count)** Concatena los elementos especificados de una matriz de cadenas, utilizando el separador especificado entre cada elemento.
23	**public int LastIndexOf (valor char)** Devuelve la posición de índice basada en cero de la última aparición del carácter Unicode especificado dentro del objeto de cadena actual.
24	**public int LastIndexOf (valor de cadena)** Devuelve la posición de índice basada en cero de la última aparición de una cadena especificada dentro del objeto de cadena actual.
25	**Eliminar cadena pública (int startIndex)** Elimina todos los caracteres de la instancia actual, comenzando en una posición específica y continuando hasta la última posición, y devuelve la cadena.
26	**Eliminar cadena pública (int startIndex, int count)** Elimina el número especificado de caracteres en la cadena actual que comienza en una posición específica y devuelve la cadena.
27	**Reemplazo de cadena pública (char oldChar, char newChar)** Reemplaza todas las apariciones de un carácter Unicode especificado en el objeto de cadena actual con el carácter Unicode especificado y devuelve la nueva cadena.
28	**cadena pública Reemplazar (cadena oldValue, cadena newValue)** Reemplaza todas las apariciones de una cadena especificada en el objeto de

	cadena actual con la cadena especificada y devuelve la nueva cadena.
29	**cadena pública [] Split (separador char params [])** Devuelve una matriz de cadenas que contiene las subcadenas en el objeto de cadena actual, delimitado por elementos de una matriz de caracteres Unicode especificada.
30	**cadena pública [] Split (separador char [], recuento int)** Devuelve una matriz de cadenas que contiene las subcadenas en el objeto de cadena actual, delimitado por elementos de una matriz de caracteres Unicode especificada. El parámetro int especifica el número máximo de subcadenas a devolver.
31	**public bool StartsWith (valor de cadena)** Determina si el comienzo de esta instancia de cadena coincide con la cadena especificada.
32	**char público [] ToCharArray ()** Devuelve una matriz de caracteres Unicode con todos los caracteres en el objeto de cadena actual.
33	**public char [] ToCharArray (int startIndex, int length)** Devuelve una matriz de caracteres Unicode con todos los caracteres en el objeto de cadena actual, comenzando desde el índice especificado y hasta la longitud especificada.
34	**cadena pública ToLower ()** Devuelve una copia de esta cadena convertida a minúsculas.
35	**cadena pública ToUpper ()** Devuelve una copia de esta cadena convertida a mayúsculas.
36	**Recorte de cadena pública ()** Elimina todos los caracteres de espacio en blanco iniciales y finales del objeto String actual.

Puede visitar la biblioteca de MSDN para obtener la lista completa de métodos y constructores de clase de cadena.

Ejemplos

El siguiente ejemplo demuestra algunos de los métodos mencionados anteriormente:

Comparar cadenas

```
using System;

namespace StringApplication {

   class StringProg {

      static void Main(string[] args) {
         string str1 = "This is test";
         string str2 = "This is text";

         if (String.Compare(str1, str2) == 0) {
            Console.WriteLine(str1 + " and " + str2 +  " are
equal.");
         } else {
            Console.WriteLine(str1 + " and " + str2 + " are
not equal.");
         }
         Console.ReadKey() ;
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
This is test and This is text are not equal.
```

Cadena contiene cadena

```
using System;

namespace StringApplication {

   class StringProg {

      static void Main(string[] args) {
         string str = "This is test";

         if (str.Contains("test")) {
            Console.WriteLine("The sequence 'test' was
found.");
         }
         Console.ReadKey() ;
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
The sequence 'test' was found.
```

Conseguir una subcadena

```
using System;

namespace StringApplication {

   class StringProg {

      static void Main(string[] args) {
         string str = "Last night I dreamt of San Pedro";
         Console.WriteLine(str);
         string substr = str.Substring(23);
         Console.WriteLine(substr);
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
San Pedro
```

Unir cuerdas

```
using System;

namespace StringApplication {

   class StringProg {

      static void Main(string[] args) {
         string[] starray = new string[]{"Down the way nights
are dark",
            "And the sun shines daily on the mountain top",
            "I took a trip on a sailing ship",
            "And when I reached Jamaica",
            "I made a stop"};

         string str = String.Join("\n", starray);
         Console.WriteLine(str);
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Down the way nights are dark
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

C # - Estructuras

En C #, una estructura es un tipo de datos de tipo de valor. Le ayuda a hacer que una sola variable contenga datos relacionados de varios tipos de datos. La palabra clave **struct** se usa para crear una estructura.

Las estructuras se utilizan para representar un registro. Supongamos que desea realizar un seguimiento de sus libros en una biblioteca. Es posible que desee realizar un seguimiento de los siguientes atributos sobre cada libro:

- Título
- Autor
- Tema
- ID del libro

Definiendo una Estructura

Para definir una estructura, debe usar la instrucción struct. La instrucción struct define un nuevo tipo de datos, con más de un miembro para su programa.

Por ejemplo, esta es la forma en que puede declarar la estructura del Libro:

```
struct Books {
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

El siguiente programa muestra el uso de la estructura:

```
using System;

struct Books {
   public string title;
   public string author;
   public string subject;
   public int book_id;
};

public class testStructure {

   public static void Main(string[] args) {
      Books Book1;   /* Declare Book1 of type Book */
      Books Book2;   /* Declare Book2 of type Book */

      /* book 1 specification */
      Book1.title = "C Programming";
      Book1.author = "Nuha Ali";
      Book1.subject = "C Programming Tutorial";
      Book1.book_id = 6495407;
```

```
      /* book 2 specification */
      Book2.title = "Telecom Billing";
      Book2.author = "Zara Ali";
      Book2.subject =  "Telecom Billing Tutorial";
      Book2.book_id = 6495700;

      /* print Book1 info */
      Console.WriteLine( "Book 1 title : {0}", Book1.title);
      Console.WriteLine("Book 1 author : {0}", Book1.author);
      Console.WriteLine("Book 1 subject : {0}",
Book1.subject);
      Console.WriteLine("Book 1 book_id :{0}",
Book1.book_id);

      /* print Book2 info */
      Console.WriteLine("Book 2 title : {0}", Book2.title);
      Console.WriteLine("Book 2 author : {0}", Book2.author);
      Console.WriteLine("Book 2 subject : {0}",
Book2.subject);
      Console.WriteLine("Book 2 book_id : {0}",
Book2.book_id);

      Console.ReadKey();
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Características de las estructuras de C

Ya ha usado una estructura simple llamada Libros. Las estructuras en C # son bastante diferentes de las de C o C ++ tradicionales. Las estructuras de C # tienen las siguientes características:

- Las estructuras pueden tener métodos, campos, indexadores, propiedades, métodos de operador y eventos.
- Las estructuras pueden tener constructores definidos, pero no destructores. Sin embargo, no puede definir un constructor predeterminado para una estructura. El constructor predeterminado se define automáticamente y no se puede cambiar.
- A diferencia de las clases, las estructuras no pueden heredar otras estructuras o clases.
- Las estructuras no se pueden usar como base para otras estructuras o clases.
- Una estructura puede implementar una o más interfaces.

- Los miembros de la estructura no pueden especificarse como abstractos, virtuales o protegidos.
- Cuando crea un objeto de estructura utilizando el operador **Nuevo** , se crea y se llama al constructor apropiado. A diferencia de las clases, las estructuras se pueden instanciar sin utilizar el operador Nuevo.
- Si no se usa el operador Nuevo, los campos permanecen sin asignar y el objeto no se puede usar hasta que se inicialicen todos los campos.

Clase versus estructura

Las clases y estructuras tienen las siguientes diferencias básicas:

- las clases son tipos de referencia y las estructuras son tipos de valor
- las estructuras no admiten herencia
- las estructuras no pueden tener un constructor predeterminado

A la luz de las discusiones anteriores, reescribamos el ejemplo anterior:

```
using System;

struct Books {
   private string title;
   private string author;
   private string subject;
   private int book_id;

   public void getValues(string t, string a, string s, int
id) {
      title = t;
      author = a;
      subject = s;
      book_id = id;
   }

   public void display() {
      Console.WriteLine("Title : {0}", title);
      Console.WriteLine("Author : {0}", author);
      Console.WriteLine("Subject : {0}", subject);
      Console.WriteLine("Book_id :{0}", book_id);
   }
};

public class testStructure {

   public static void Main(string[] args) {
      Books Book1 = new Books();   /* Declare Book1 of type
Book */
      Books Book2 = new Books();   /* Declare Book2 of type
Book */

      /* book 1 specification */
      Book1.getValues("C Programming",
```

```
      "Nuha Ali", "C Programming Tutorial",6495407);

      /* book 2 specification */
      Book2.getValues("Telecom Billing",
      "Zara Ali", "Telecom Billing Tutorial", 6495700);

      /* print Book1 info */
      Book1.display();

      /* print Book2 info */
      Book2.display();

      Console.ReadKey();
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

C # - Enums

Una enumeración es un conjunto de constantes enteras con nombre. Un tipo enumerado se declara utilizando la palabra clave **enum** .

Las enumeraciones de C # son tipos de datos de valor. En otras palabras, la enumeración contiene sus propios valores y no puede heredar o no puede heredar.

Declarando *enum* Variable

La sintaxis general para declarar una enumeración es:

```
enum <enum_name> {
   enumeration list
};
```

Dónde,

- El *enum_name* especifica el nombre del tipo de enumeración.
- La *lista de enumeración* es una lista de identificadores separados por comas.

Cada uno de los símbolos en la lista de enumeración representa un valor entero, uno mayor que el símbolo que lo precede. Por defecto, el valor del primer símbolo de enumeración es 0. Por ejemplo:

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

Ejemplo

El siguiente ejemplo demuestra el uso de la variable enum:

```
using System;

namespace EnumApplication {

   class EnumProgram {
      enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

      static void Main(string[] args) {
         int WeekdayStart = (int)Days.Mon;
         int WeekdayEnd = (int)Days.Fri;
         Console.WriteLine("Monday: {0}", WeekdayStart);
         Console.WriteLine("Friday: {0}", WeekdayEnd);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Monday: 1
Friday: 5
```

C # - Clases

Cuando define una clase, define un plano para un tipo de datos. En realidad, esto no define ningún dato, pero sí define lo que significa el nombre de la clase. Es decir, en qué consiste un objeto de la clase y qué operaciones se pueden realizar en ese objeto. Los objetos son instancias de una clase. Los métodos y variables que constituyen una clase se llaman miembros de la clase.

Definiendo una clase

Una definición de clase comienza con la clase de palabra clave seguida del nombre de la clase; y el cuerpo de la clase encerrado por un par de llaves. A continuación se presenta la forma general de una definición de clase:

```
<access specifier> class  class_name {
   // member variables
   <access specifier> <data type> variable1;
   <access specifier> <data type> variable2;
   ...
   <access specifier> <data type> variableN;
   // member methods
   <access specifier> <return type> method1(parameter_list) {
      // method body
   }
   <access specifier> <return type> method2(parameter_list) {
      // method body
```

```
    }
    ...
    <access specifier> <return type> methodN(parameter_list) {
       // method body
    }
}
```

Nota -

- Los especificadores de acceso especifican las reglas de acceso para los miembros, así como para la clase misma. Si no se menciona, el especificador de acceso predeterminado para un tipo de clase es **interno** . El acceso predeterminado para los miembros es **privado** .
- El tipo de datos especifica el tipo de variable, y el tipo de retorno especifica el tipo de datos de los datos que devuelve el método, si corresponde.
- Para acceder a los miembros de la clase, utilice el operador de punto (.).
- El operador de punto vincula el nombre de un objeto con el nombre de un miembro.

El siguiente ejemplo ilustra los conceptos discutidos hasta ahora:

```
using System;

namespace BoxApplication {

    class Box {
       public double length;   // Length of a box
       public double breadth;  // Breadth of a box
       public double height;   // Height of a box
    }

    class Boxtester {

        static void Main(string[] args) {
            Box Box1 = new Box();   // Declare Box1 of type
Box
            Box Box2 = new Box();   // Declare Box2 of type
Box
            double volume = 0.0;    // Store the volume of a
box here

            // box 1 specification
            Box1.height = 5.0;
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // box 2 specification
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // volume of box 1
            volume = Box1.height * Box1.length *
Box1.breadth;
```

```
            Console.WriteLine("Volume of Box1 : {0}",
volume);

            // volume of box 2
            volume = Box2.height * Box2.length *
Box2.breadth;
            Console.WriteLine("Volume of Box2 : {0}",
volume);
            Console.ReadKey();
         }
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Funciones de miembros y encapsulación

Una función miembro de una clase es una función que tiene su definición o su prototipo dentro de la definición de clase similar a cualquier otra variable. Opera en cualquier objeto de la clase de la que es miembro, y tiene acceso a todos los miembros de una clase para ese objeto.

Las variables miembro son los atributos de un objeto (desde la perspectiva del diseño) y se mantienen privadas para implementar la encapsulación. Solo se puede acceder a estas variables mediante las funciones de miembro público.

Pongamos los conceptos anteriores para establecer y obtener el valor de diferentes miembros de una clase en una clase:

```
using System;

namespace BoxApplication {

   class Box {
      private double length;   // Length of a box
      private double breadth;  // Breadth of a box
      private double height;   // Height of a box

      public void setLength( double len ) {
         length = len;
      }

      public void setBreadth( double bre ) {
         breadth = bre;
      }

      public void setHeight( double hei ) {
         height = hei;
      }
      public double getVolume() {
         return length * breadth * height;
```

```
      }
   }

   class Boxtester {

      static void Main(string[] args) {
         Box Box1 = new Box();   // Declare Box1 of type Box
         Box Box2 = new Box();
         double volume;

         // Declare Box2 of type Box
         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);

         // volume of box 1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}" ,volume);

         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Constructores C

Un **constructor de** clase es una función miembro especial de una clase que se ejecuta cada vez que creamos nuevos objetos de esa clase.

Un constructor tiene exactamente el mismo nombre que el de clase y no tiene ningún tipo de retorno. El siguiente ejemplo explica el concepto de constructor:

```
using System;

namespace LineApplication {

   class Line {
      private double length;   // Length of a line
```

```
      public Line() {
         Console.WriteLine("Object is being created");
      }

      public void setLength( double len ) {
         length = len;
      }

      public double getLength() {
         return length;
      }

      static void Main(string[] args) {
         Line line = new Line();

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}",
line.getLength());
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Object is being created
Length of line : 6
```

Un **constructor predeterminado** no tiene ningún parámetro, pero si lo necesita, un constructor puede tener parámetros. Dichos constructores se denominan **constructores parametrizados** . Esta técnica le ayuda a asignar un valor inicial a un objeto en el momento de su creación, como se muestra en el siguiente ejemplo:

```
using System;

namespace LineApplication {

   class Line {
      private double length;   // Length of a line

      public Line(double len) {  //Parameterized constructor
         Console.WriteLine("Object is being created, length =
{0}", len);
         length = len;
      }

      public void setLength( double len ) {
         length = len;
      }

      public double getLength() {
         return length;
```

```
      }

      static void Main(string[] args) {
         Line line = new Line(10.0);
         Console.WriteLine("Length of line : {0}",
line.getLength());

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}",
line.getLength());
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Destructores C

Un **destructor** es una función miembro especial de una clase que se ejecuta cada vez que un objeto de su clase queda fuera de alcance. Un **destructor** tiene exactamente el mismo nombre que el de la clase con una tilde prefijada (~) y no puede devolver un valor ni puede tomar ningún parámetro.

Destructor puede ser muy útil para liberar recursos de memoria antes de salir del programa. Los destructores no se pueden heredar ni sobrecargar.

El siguiente ejemplo explica el concepto de destructor:

```
using System;

namespace LineApplication {

   class Line {
      private double length;   // Length of a line

      public Line() {   // constructor
         Console.WriteLine("Object is being created");
      }

      ~Line() {   //destructor
         Console.WriteLine("Object is being deleted");
      }

      public void setLength( double len ) {
         length = len;
      }
```

```
      public double getLength() {
         return length;
      }

      static void Main(string[] args) {
         Line line = new Line();

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}",
line.getLength());
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Object is being created
Length of line : 6
Object is being deleted
```

Miembros estáticos de una clase C

Podemos definir miembros de la clase como estáticos usando la palabra clave **estática** . Cuando declaramos un miembro de una clase como estático, significa que no importa cuántos objetos de la clase se creen, solo hay una copia del miembro estático.

La palabra clave **static** implica que solo existe una instancia del miembro para una clase. Las variables estáticas se utilizan para definir constantes porque sus valores se pueden recuperar invocando la clase sin crear una instancia de la misma. Las variables estáticas se pueden inicializar fuera de la función miembro o la definición de clase. También puede inicializar variables estáticas dentro de la definición de clase.

El siguiente ejemplo demuestra el uso de **variables estáticas** :

```
using System;

namespace StaticVarApplication {

   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }

      public int getNum() {
         return num;
      }
   }

   class StaticTester {
```

```
      static void Main(string[] args) {
         StaticVar s1 = new StaticVar();
         StaticVar s2 = new StaticVar();
         s1.count();
         s1.count();
         s1.count();
         s2.count();
         s2.count();
         s2.count();
         Console.WriteLine("Variable num for s1: {0}",
s1.getNum());
         Console.WriteLine("Variable num for s2: {0}",
s2.getNum());
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Variable num for s1: 6
Variable num for s2: 6
```

También puede declarar una **función miembro** como **estática** . Dichas funciones solo pueden acceder a variables estáticas. Las funciones estáticas existen incluso antes de que se cree el objeto. El siguiente ejemplo demuestra el uso de **funciones estáticas** :

```
using System;

namespace StaticVarApplication {

   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }

      public static int getNum() {
         return num;
      }
   }

   class StaticTester {

      static void Main(string[] args) {
         StaticVar s = new StaticVar();
         s.count();
         s.count();
         s.count();
         Console.WriteLine("Variable num: {0}",
StaticVar.getNum());
```

```
            Console.ReadKey();
        }
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Variable num: 3
```

C # - Herencia

Uno de los conceptos más importantes en la programación orientada a objetos es la herencia. La herencia nos permite definir una clase en términos de otra clase, lo que facilita la creación y el mantenimiento de una aplicación. Esto también brinda la oportunidad de reutilizar la funcionalidad del código y acelera el tiempo de implementación.

Al crear una clase, en lugar de escribir miembros de datos y funciones de miembros completamente nuevos, el programador puede designar que la nueva clase herede los miembros de una clase existente. Esta clase existente se denomina clase **base** , y la nueva clase se conoce como la clase **derivada** .

La idea de herencia implementa la relación **IS-A** . Por ejemplo, el mamífero **es un** animal, el perro **es un** mamífero, por lo tanto, el perro **es un** animal también, y así sucesivamente.

Clases Base y Derivadas

Una clase puede derivarse de más de una clase o interfaz, lo que significa que puede heredar datos y funciones de múltiples clases o interfaces base.

La sintaxis utilizada en C # para crear clases derivadas es la siguiente:

```
<acess-specifier> class <base_class> {
   ...
}

class <derived_class> : <base_class> {
   ...
}
```

Considere una clase base Shape y su clase derivada Rectángulo -

```
using System;

namespace InheritanceApplication {

   class Shape {

      public void setWidth(int w) {
         width = w;
      }

      public void setHeight(int h) {
```

```
            height = h;
         }
         protected int width;
         protected int height;
      }

      // Derived class
      class Rectangle: Shape {

         public int getArea() {
            return (width * height);
         }
      }

      class RectangleTester {

         static void Main(string[] args) {
            Rectangle Rect = new Rectangle();

            Rect.setWidth(5);
            Rect.setHeight(7);

            // Print the area of the object.
            Console.WriteLine("Total area: {0}",
Rect.getArea());
            Console.ReadKey();
         }
      }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Total area: 35
```

Inicializando clase base

La clase derivada hereda las variables miembro y los métodos miembro de la clase base. Por lo tanto, el objeto de superclase debe crearse antes de crear la subclase. Puede dar instrucciones para la inicialización de la superclase en la lista de inicialización de miembros.

El siguiente programa demuestra esto:

```
using System;

namespace RectangleApplication {

   class Rectangle {
      //member variables
      protected double length;
      protected double width;

      public Rectangle(double l, double w) {
```

```
         length = l;
         width = w;
      }

      public double GetArea() {
         return length * width;
      }

      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle

   class Tabletop : Rectangle {
      private double cost;
      public Tabletop(double l, double w) : base(l, w) { }

      public double GetCost() {
         double cost;
         cost = GetArea() * 70;
         return cost;
      }

      public void Display() {
         base.Display();
         Console.WriteLine("Cost: {0}", GetCost());
      }
   }

   class ExecuteRectangle {

      static void Main(string[] args) {
         Tabletop t = new Tabletop(4.5, 7.5);
         t.Display();
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

Herencia múltiple en C

C # no admite herencia múltiple . Sin embargo, puede usar interfaces para implementar herencia múltiple. El siguiente programa demuestra esto:

```
using System;

namespace InheritanceApplication {

   class Shape {

      public void setWidth(int w) {
         width = w;
      }

      public void setHeight(int h) {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Base class PaintCost
   public interface PaintCost {
      int getCost(int area);
   }

   // Derived class
   class Rectangle : Shape, PaintCost {

      public int getArea() {
         return (width * height);
      }

      public int getCost(int area) {
         return area * 70;
      }
   }

   class RectangleTester {

      static void Main(string[] args) {
         Rectangle Rect = new Rectangle();
         int area;
         Rect.setWidth(5);
         Rect.setHeight(7);
         area = Rect.getArea();

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",
Rect.getArea());
         Console.WriteLine("Total paint cost: ${0}" ,
Rect.getCost(area));
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Total area: 35
Total paint cost: $2450
```

C # - Polimorfismo

La palabra **polimorfismo** significa tener muchas formas. En el paradigma de programación orientado a objetos, el polimorfismo a menudo se expresa como 'una interfaz, múltiples funciones'.

El polimorfismo puede ser estático o dinámico. En **el polimorfismo estático** , la respuesta a una función se determina en el momento de la compilación. En **el polimorfismo dinámico** , se decide en tiempo de ejecución.

Polimorfismo estático

El mecanismo de vinculación de una función con un objeto durante el tiempo de compilación se denomina enlace temprano. También se llama enlace estático. C # proporciona dos técnicas para implementar el polimorfismo estático. Ellos son

- Función de sobrecarga
- Sobrecarga del operador

Discutimos sobrecarga del operador en el próximo capítulo.

Sobrecarga de funciones

Puede tener varias definiciones para el mismo nombre de función en el mismo ámbito. La definición de la función debe diferir entre sí por los tipos y / o el número de argumentos en la lista de argumentos. No puede sobrecargar las declaraciones de funciones que difieren solo por tipo de retorno.

El siguiente ejemplo muestra el uso de la función **print ()** para imprimir diferentes tipos de datos:

```
using System;

namespace PolymorphismApplication {

   class Printdata {

      void print(int i) {
         Console.WriteLine("Printing int: {0}", i );
      }

      void print(double f) {
         Console.WriteLine("Printing float: {0}" , f);
      }

      void print(string s) {
```

```
         Console.WriteLine("Printing string: {0}", s);
      }

      static void Main(string[] args) {
         Printdata p = new Printdata();

         // Call print to print integer
         p.print(5);

         // Call print to print float
         p.print(500.263);

         // Call print to print string
         p.print("Hello C++");
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

Polimorfismo Dinámico

C # le permite crear clases abstractas que se utilizan para proporcionar una implementación de clase parcial de una interfaz. La implementación se completa cuando una clase derivada hereda de ella. **Las** clases abstractas contienen métodos abstractos, que son implementados por la clase derivada. Las clases derivadas tienen una funcionalidad más especializada.

Aquí están las reglas sobre las clases abstractas:

- No puede crear una instancia de una clase abstracta
- No puede declarar un método abstracto fuera de una clase abstracta
- Cuando una clase se declara **sellada** , no se puede heredar, las clases abstractas no se pueden declarar selladas.

El siguiente programa muestra una clase abstracta:

```
using System;

namespace PolymorphismApplication {

   abstract class Shape {
      public abstract int area();
   }

   class Rectangle:  Shape {
      private int length;
      private int width;
```

```
      public Rectangle( int a = 0, int b = 0) {
         length = a;
         width = b;
      }

      public override int area () {
         Console.WriteLine("Rectangle class area :");
         return (width * length);
      }
   }

   class RectangleTester {

      static void Main(string[] args) {
         Rectangle r = new Rectangle(10, 7);
         double a = r.area();
         Console.WriteLine("Area: {0}",a);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Rectangle class area :
Area: 70
```

Cuando tiene una función definida en una clase que desea implementar en una clase o clases heredadas, utiliza funciones **virtuales** . Las funciones virtuales podrían implementarse de manera diferente en diferentes clases heredadas y la llamada a estas funciones se decidirá en tiempo de ejecución.

El polimorfismo dinámico se implementa mediante **clases abstractas** y **funciones virtuales** .

El siguiente programa demuestra esto:

```
using System;

namespace PolymorphismApplication {

   class Shape {
      protected int width, height;

      public Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      public virtual int area() {
         Console.WriteLine("Parent class area :");
         return 0;
      }
   }
```

```
   class Rectangle: Shape {
      public Rectangle( int a = 0, int b = 0): base(a, b) {

      }

      public override int area () {
         Console.WriteLine("Rectangle class area :");
         return (width * height);
      }
   }

   class Triangle: Shape {
      public Triangle(int a = 0, int b = 0): base(a, b) {

      }

      public override int area() {
         Console.WriteLine("Triangle class area :");
         return (width * height / 2);
      }
   }

   class Caller {
      public void CallArea(Shape sh) {
         int a;
         a = sh.area();
         Console.WriteLine("Area: {0}", a);
      }
   }

   class Tester {

      static void Main(string[] args) {
         Caller c = new Caller();
         Rectangle r = new Rectangle(10, 7);
         Triangle t = new Triangle(10, 5);
         c.CallArea(r);
         c.CallArea(t);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

C # - Sobrecarga del operador

Puede redefinir o sobrecargar la mayoría de los operadores integrados disponibles en C #. Por lo tanto, un programador puede usar operadores con tipos definidos por el usuario también. Los operadores sobrecargados son funciones con nombres especiales: **operador de** palabra clave seguido del símbolo del operador que se está definiendo. similar a cualquier otra función, un operador sobrecargado tiene un tipo de retorno y una lista de parámetros.

Por ejemplo, realice la siguiente función:

```
public static Box operator+ (Box b, Box c) {
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

La función anterior implementa el operador de suma (+) para un cuadro de clase definido por el usuario. Agrega los atributos de dos objetos Box y devuelve el objeto Box resultante.

Implementación de la sobrecarga del operador

El siguiente programa muestra la implementación completa:

```
using System;

namespace OperatorOvlApplication {

   class Box {
      private double length;   // Length of a box
      private double breadth;  // Breadth of a box
      private double height;   // Height of a box

      public double getVolume() {
         return length * breadth * height;
      }

      public void setLength( double len ) {
         length = len;
      }

      public void setBreadth( double bre ) {
         breadth = bre;
      }

      public void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      public static Box operator+ (Box b, Box c) {
         Box box = new Box();
```

```
         box.length = b.length + c.length;
         box.breadth = b.breadth + c.breadth;
         box.height = b.height + c.height;
         return box;
      }
   }

   class Tester {

      static void Main(string[] args) {
         Box Box1 = new Box();   // Declare Box1 of type Box
         Box Box2 = new Box();   // Declare Box2 of type Box
         Box Box3 = new Box();   // Declare Box3 of type Box
         double volume = 0.0;    // Store the volume of a box
here

         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);

         // volume of box 1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}", volume);

         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         // Add two object as follows:
         Box3 = Box1 + Box2;

         // volume of box 3
         volume = Box3.getVolume();
         Console.WriteLine("Volume of Box3 : {0}", volume);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Operadores sobrecargables y no sobrecargables

La siguiente tabla describe la capacidad de sobrecarga de los operadores en C # -

No Señor.	Operadores y Descripción
1	**+, -,!, ~, ++, -** Estos operadores unarios toman un operando y pueden sobrecargarse.
2	**+, -, *, /,%** Estos operadores binarios toman un operando y pueden sobrecargarse.
3	**==,! =, <,>, <=,> =** Los operadores de comparación se pueden sobrecargar.
4 4	**&&, ||** Los operadores lógicos condicionales no se pueden sobrecargar directamente.
5 5	**+ =, - =, * =, / =,% =** Los operadores de asignación no se pueden sobrecargar.
6 6	**=,.,?:, ->, nuevo, es, sizeof, typeof** Estos operadores no se pueden sobrecargar.

Ejemplo

A la luz de las discusiones anteriores, ampliemos el ejemplo anterior y sobrecarguemos algunos operadores más:

```
using System;

namespace OperatorOvlApplication {

   class Box {
      private double length;     // Length of a box
      private double breadth;    // Breadth of a box
      private double height;     // Height of a box

      public double getVolume() {
         return length * breadth * height;
      }

      public void setLength( double len ) {
```

```
         length = len;
      }

      public void setBreadth( double bre ) {
         breadth = bre;
      }

      public void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      public static Box operator+ (Box b, Box c) {
         Box box = new Box();
         box.length = b.length + c.length;
         box.breadth = b.breadth + c.breadth;
         box.height = b.height + c.height;
         return box;
      }

      public static bool operator == (Box lhs, Box rhs) {
         bool status = false;
         if (lhs.length == rhs.length && lhs.height ==
rhs.height && lhs.breadth == rhs.breadth) {
            status = true;
         }
         return status;
      }

      public static bool operator !=(Box lhs, Box rhs) {
         bool status = false;

         if (lhs.length != rhs.length || lhs.height !=
rhs.height || lhs.breadth != rhs.breadth) {
            status = true;
         }
         return status;
      }

      public static bool operator <(Box lhs, Box rhs) {
         bool status = false;

         if (lhs.length < rhs.length && lhs.height <
rhs.height && lhs.breadth < rhs.breadth) {
            status = true;
         }
         return status;
      }

      public static bool operator >(Box lhs, Box rhs) {
         bool status = false;
```

```
         if (lhs.length > rhs.length && lhs.height >
rhs.height && lhs.breadth > rhs.breadth) {
            status = true;
         }
         return status;
      }

      public static bool operator <=(Box lhs, Box rhs) {
         bool status = false;

         if (lhs.length <= rhs.length && lhs.height <=
rhs.height && lhs.breadth <= rhs.breadth) {
            status = true;
         }
         return status;
      }

      public static bool operator >=(Box lhs, Box rhs) {
         bool status = false;

         if (lhs.length >= rhs.length && lhs.height >=
rhs.height && lhs.breadth >= rhs.breadth) {
            status = true;
         }
         return status;
      }

      public override string ToString() {
         return String.Format("({0}, {1}, {2})", length,
breadth, height);
      }
   }

   class Tester {

      static void Main(string[] args) {
         Box Box1 = new Box();   // Declare Box1 of type Box
         Box Box2 = new Box();   // Declare Box2 of type Box
         Box Box3 = new Box();   // Declare Box3 of type Box
         Box Box4 = new Box();
         double volume = 0.0;    // Store the volume of a box
here

         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);
```

```
         //displaying the Boxes using the overloaded
ToString():
         Console.WriteLine("Box 1: {0}", Box1.ToString());
         Console.WriteLine("Box 2: {0}", Box2.ToString());

         // volume of box 1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}", volume);

         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         // Add two object as follows:
         Box3 = Box1 + Box2;
         Console.WriteLine("Box 3: {0}", Box3.ToString());

         // volume of box 3
         volume = Box3.getVolume();
         Console.WriteLine("Volume of Box3 : {0}", volume);

         //comparing the boxes
         if (Box1 > Box2)
            Console.WriteLine("Box1 is greater than Box2");
         else
            Console.WriteLine("Box1 is  greater than Box2");

         if (Box1 < Box2)
            Console.WriteLine("Box1 is less than Box2");
         else
            Console.WriteLine("Box1 is not less than Box2");

         if (Box1 >= Box2)
            Console.WriteLine("Box1 is greater or equal to
Box2");
         else
            Console.WriteLine("Box1 is not greater or equal
to Box2");

         if (Box1 <= Box2)
            Console.WriteLine("Box1 is less or equal to
Box2");
         else
            Console.WriteLine("Box1 is not less or equal to
Box2");

         if (Box1 != Box2)
            Console.WriteLine("Box1 is not equal to Box2");
         else
            Console.WriteLine("Box1 is not greater or equal
to Box2");
         Box4 = Box3;
```

```
            if (Box3 == Box4)
               Console.WriteLine("Box3 is equal to Box4");
            else
               Console.WriteLine("Box3 is not equal to Box4");

            Console.ReadKey();
         }
      }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Box 1: (6, 7, 5)
Box 2: (12, 13, 10)
Volume of Box1 : 210
Volume of Box2 : 1560
Box 3: (18, 20, 15)
Volume of Box3 : 5400
Box1 is not greater than Box2
Box1 is less than Box2
Box1 is not greater or equal to Box2
Box1 is less or equal to Box2
Box1 is not equal to Box2
Box3 is equal to Box4
```

C # - Interfaces

Una interfaz se define como un contrato sintáctico que todas las clases que heredan la interfaz deben seguir. La interfaz define la parte **'qué'** del contrato sintáctico y las clases derivadas definen la parte **'cómo'** del contrato sintáctico.

Las interfaces definen propiedades, métodos y eventos, que son los miembros de la interfaz. Las interfaces contienen solo la declaración de los miembros. Es responsabilidad de la clase derivada definir los miembros. A menudo ayuda a proporcionar una estructura estándar que seguirían las clases derivadas.

Las clases abstractas, en cierta medida, tienen el mismo propósito, sin embargo, se usan principalmente cuando solo unos pocos métodos deben ser declarados por la clase base y la clase derivada implementa las funcionalidades.

Declarando interfaces

Las interfaces se declaran utilizando la palabra clave de interfaz. Es similar a la declaración de clase. Las declaraciones de interfaz son públicas por defecto. El siguiente es un ejemplo de una declaración de interfaz:

```
public interface ITransactions {
   // interface members
   void showTransaction();
   double getAmount();
}
```

Ejemplo

El siguiente ejemplo demuestra la implementación de la interfaz anterior:

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System;

namespace InterfaceApplication {

   public interface ITransactions {
      // interface members
      void showTransaction();
      double getAmount();
   }

   public class Transaction : ITransactions {
      private string tCode;
      private string date;
      private double amount;

      public Transaction() {
         tCode = " ";
         date = " ";
         amount = 0.0;
      }

      public Transaction(string c, string d, double a) {
         tCode = c;
         date = d;
         amount = a;
      }

      public double getAmount() {
         return amount;
      }

      public void showTransaction() {
         Console.WriteLine("Transaction: {0}", tCode);
         Console.WriteLine("Date: {0}", date);
         Console.WriteLine("Amount: {0}", getAmount());
      }
   }

   class Tester {

      static void Main(string[] args) {
         Transaction t1 = new Transaction("001", "8/10/2012",
78900.00);
         Transaction t2 = new Transaction("002", "9/10/2012",
451900.00);
         t1.showTransaction();
```

```
            t2.showTransaction();
            Console.ReadKey();
        }
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

C # - Espacios de nombres

Un **espacio de nombres** está diseñado para proporcionar una manera de mantener un conjunto de nombres separado de otro. Los nombres de clase declarados en un espacio de nombres no entran en conflicto con los mismos nombres de clase declarados en otro.

Definiendo un espacio de nombres

Una definición de espacio de nombres comienza con el **espacio de nombres de** palabras clave seguido del nombre del espacio de nombres de la siguiente manera:

```
namespace namespace_name {
   // code declarations
}
```

Para llamar a la versión habilitada para el espacio de nombres de una función o variable, anteponga el nombre del espacio de nombres de la siguiente manera:

```
namespace_name.item_name;
```

El siguiente programa demuestra el uso de espacios de nombres:

```
using System;

namespace first_space {

   class namespace_cl {

      public void func() {
         Console.WriteLine("Inside first_space");
      }
   }
}

namespace second_space {

   class namespace_cl {
```

```
      public void func() {
         Console.WriteLine("Inside second_space");
      }
   }
}

class TestClass {

   static void Main(string[] args) {
      first_space.namespace_cl fc = new
first_space.namespace_cl();
      second_space.namespace_cl sc = new
second_space.namespace_cl();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Inside first_space
Inside second_space
```

El *uso de* palabras clave

La palabra clave **using** indica que el programa está usando los nombres en el espacio de nombres dado. Por ejemplo, estamos usando el espacio de nombres del **sistema** en nuestros programas. La clase Console se define allí. Solo escribimos

```
Console.WriteLine ("Hello there");
```

Podríamos haber escrito el nombre completo como:

```
System.Console.WriteLine("Hello there");
```

También puede evitar anteponer espacios de nombres con la directiva **using** space. Esta directiva le dice al compilador que el código subsiguiente está haciendo uso de nombres en el espacio de nombres especificado. El espacio de nombres está implícito para el siguiente código:

Reescribamos nuestro ejemplo anterior, con el uso de la directiva:

```
using System;
using first_space;
using second_space;

namespace first_space {

   class abc {

      public void func() {
```

```
         Console.WriteLine("Inside first_space");
      }
   }
}

namespace second_space {

   class efg {

      public void func() {
         Console.WriteLine("Inside second_space");
      }
   }
}

class TestClass {

   static void Main(string[] args) {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Inside first_space
Inside second_space
```

Espacios de nombres anidados

Puede definir un espacio de nombres dentro de otro espacio de nombres de la siguiente manera:

```
namespace namespace_name1 {

   // code declarations
   namespace namespace_name2 {
      // code declarations
   }
}
```

Puede acceder a los miembros del espacio de nombres anidado utilizando el operador de punto (.) De la siguiente manera:

```
using System;
using first_space;
using first_space.second_space;

namespace first_space {

   class abc {
```

```
      public void func() {
         Console.WriteLine("Inside first_space");
      }
   }

   namespace second_space {

      class efg {

         public void func() {
            Console.WriteLine("Inside second_space");
         }
      }
   }
}

class TestClass {

   static void Main(string[] args) {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Inside first_space
Inside second_space
```

C # - Directivas de preprocesador

Las directivas de preprocesador dan instrucciones al compilador para preprocesar la información antes de que comience la compilación real.

Todas las directivas de preprocesador comienzan con #, y solo los caracteres de espacio en blanco pueden aparecer antes de una directiva de preprocesador en una línea. Las directivas de preprocesador no son declaraciones, por lo que no terminan con punto y coma (;).

El compilador de C # no tiene un preprocesador separado; sin embargo, las directivas se procesan como si hubiera una. En C #, las directivas de preprocesador se utilizan para ayudar en la compilación condicional. A diferencia de las directivas C y C ++, no se utilizan para crear macros. Una directiva de preprocesador debe ser la única instrucción en una línea.

Directivas de preprocesador en C

La siguiente tabla enumera las directivas de preprocesador disponibles en C # -

No Señor.	Directiva y descripción del preprocesador
1	**#definir** Define una secuencia de caracteres, llamada símbolo.
2	**#undef** Le permite definir un símbolo.
3	**#Si** Permite probar un símbolo o símbolos para ver si se evalúan como verdaderos.
4 4	**#más** Permite crear una directiva condicional compuesta, junto con #if.
5 5	**#elif** Permite crear una directiva condicional compuesta.
6 6	**#terminara si** Especifica el final de una directiva condicional.
7 7	**#línea** Le permite modificar el número de línea del compilador y (opcionalmente) la salida del nombre del archivo para errores y advertencias.
8	**#error** Permite generar un error desde una ubicación específica en su código.
9 9	**#advertencia** Permite generar una advertencia de nivel uno desde una ubicación específica en su código.
10	**#región** Le permite especificar un bloque de código que puede expandir o contraer al usar la función de esquema del Editor de código de Visual Studio.
11	**#endregion**

	Marca el final de un bloque #region.

El preprocesador #define

La directiva #define preprocessor crea constantes simbólicas.

#define le permite definir un símbolo de modo que, al usar el símbolo como la expresión pasada a la directiva #if, la expresión se evalúe como verdadera. Su sintaxis es la siguiente:

```
#define symbol
```

El siguiente programa ilustra esto:

```
#define PI
using System;

namespace PreprocessorDAppl {

   class Program {

      static void Main(string[] args) {
         #if (PI)
            Console.WriteLine("PI is defined");
         #else
            Console.WriteLine("PI is not defined");
         #endif
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
PI is defined
```

Directivas condicionales

Puede usar la directiva #if para crear una directiva condicional. Las directivas condicionales son útiles para probar un símbolo o símbolos para verificar si se evalúan como verdaderos. Si se evalúan como verdaderos, el compilador evalúa todo el código entre el #if y la siguiente directiva.

La sintaxis para la directiva condicional es -

```
#if symbol [operator symbol]...
```

Donde, *símbolo* es el nombre del símbolo que desea probar. También puede usar verdadero y falso o anteponer el símbolo con el operador de negación.

El *símbolo* del operador es el operador utilizado para evaluar el símbolo. Los operadores podrían ser cualquiera de los siguientes:

- == (igualdad)

- ! = (desigualdad)
- && (y)
- || (o)

También puede agrupar símbolos y operadores con paréntesis. Las directivas condicionales se usan para compilar código para una compilación de depuración o al compilar para una configuración específica. Una directiva condicional que comienza con una directiva **#if** debe terminarse explícitamente con una directiva **#endif** .

El siguiente programa demuestra el uso de directivas condicionales:

```
#define DEBUG
#define VC_V10
using System;

public class TestClass {

   public static void Main() {
      #if (DEBUG && !VC_V10)
         Console.WriteLine("DEBUG is defined");
      #elif (!DEBUG && VC_V10)
         Console.WriteLine("VC_V10 is defined");
      #elif (DEBUG && VC_V10)
         Console.WriteLine("DEBUG and VC_V10 are defined");
      #else
         Console.WriteLine("DEBUG and VC_V10 are not
defined");
      #endif
      Console.ReadKey();
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
DEBUG and VC_V10 are defined
```

C # - Expresiones regulares

Una **expresión regular** es un patrón que podría compararse con un texto de entrada. El marco .Net proporciona un motor de expresión regular que permite dicha coincidencia. Un patrón consta de uno o más literales de caracteres, operadores o construcciones.

Construcciones para definir expresiones regulares

Existen varias categorías de caracteres, operadores y construcciones que le permiten definir expresiones regulares. Haga clic en los siguientes enlaces para encontrar estas construcciones.

- Escapes de personajes
- Clases de personajes
- Anclas

- Construcciones de agrupamiento
- Cuantificadores
- Construcciones de referencia
- Construcciones de alternancia
- Sustituciones
- Construcciones misceláneas

La clase de expresiones regulares

La clase Regex se usa para representar una expresión regular. Tiene los siguientes métodos de uso común:

No Señor.	Métodos y descripción
1	**public bool IsMatch (entrada de cadena)** Indica si la expresión regular especificada en el constructor Regex encuentra una coincidencia en una cadena de entrada especificada.
2	**public bool IsMatch (entrada de cadena, int startat)** Indica si la expresión regular especificada en el constructor Regex encuentra una coincidencia en la cadena de entrada especificada, comenzando en la posición inicial especificada en la cadena.
3	**public static bool IsMatch (entrada de cadena, patrón de cadena)** Indica si la expresión regular especificada encuentra una coincidencia en la cadena de entrada especificada.
4 4	**Partidos públicos de MatchCollection (entrada de cadena)** Busca en la cadena de entrada especificada todas las apariciones de una expresión regular.
5 5	**Reemplazo de cadena pública (entrada de cadena, reemplazo de cadena)** En una cadena de entrada especificada, reemplaza todas las cadenas que coinciden con un patrón de expresión regular con una cadena de reemplazo especificada.
6 6	**cadena pública [] División (entrada de cadena)** Divide una cadena de entrada en una matriz de subcadenas en las posiciones definidas por un patrón de expresión regular especificado en el constructor Regex.

Para obtener la lista completa de métodos y propiedades, lea la documentación de Microsoft en C #.

Ejemplo 1

El siguiente ejemplo coincide con palabras que comienzan con 'S':

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication {

   class Program {

      private static void showMatch(string text, string expr)
{
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc) {
            Console.WriteLine(m);
         }
      }

      static void Main(string[] args) {
         string str = "A Thousand Splendid Suns";

         Console.WriteLine("Matching words that start with
'S': ");
         showMatch(str, @"\bS\S*");
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Matching words that start with 'S':
The Expression: \bS\S*
Splendid
Suns
```

Ejemplo 2

El siguiente ejemplo coincide con palabras que comienzan con 'm' y terminan con 'e':

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication {

   class Program {
```

```
      private static void showMatch(string text, string expr)
{
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc) {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args) {
         string str = "make maze and manage to measure it";

         Console.WriteLine("Matching words start with 'm' and
ends with 'e':");
         showMatch(str, @"\bm\S*e\b");
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Matching words start with 'm' and ends with 'e':
The Expression: \bm\S*e\b
make
maze
manage
measure
```

Ejemplo 3

Este ejemplo reemplaza el espacio en blanco adicional:

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication {

   class Program {

      static void Main(string[] args) {
         string input = "Hello   World   ";
         string pattern = "\\s+";
         string replacement = " ";
         Regex rgx = new Regex(pattern);
         string result = rgx.Replace(input, replacement);

         Console.WriteLine("Original String: {0}", input);
         Console.WriteLine("Replacement String: {0}",
result);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Original String: Hello World
Replacement String: Hello World
```

C # - Manejo de excepciones

Una excepción es un problema que surge durante la ejecución de un programa. La excepción AC # es una respuesta a una circunstancia excepcional que surge mientras se ejecuta un programa, como un intento de dividir por cero.

Las excepciones proporcionan una forma de transferir el control de una parte de un programa a otra. El manejo de excepciones de C # se basa en cuatro palabras clave: **probar** , **atrapar** , **finalmente** y **lanzar** .

- **try** : un bloque try identifica un bloque de código para el cual se activan excepciones particulares. Es seguido por uno o más bloques de captura.
- **catch** : un programa detecta una excepción con un controlador de excepciones en el lugar del programa donde desea manejar el problema. La palabra clave catch indica la captura de una excepción.
- **finally** : el bloque finally se usa para ejecutar un conjunto dado de instrucciones, ya sea que se lance una excepción o no. Por ejemplo, si abre un archivo, debe cerrarse si se genera una excepción o no.
- **throw** : un programa genera una excepción cuando aparece un problema. Esto se hace usando una palabra clave throw.

Sintaxis

Suponiendo que un bloque genera una excepción, un método detecta una excepción utilizando una combinación de las palabras clave try y catch. Se coloca un bloque try / catch alrededor del código que podría generar una excepción. El código dentro de un bloque try / catch se conoce como código protegido, y la sintaxis para usar try / catch es similar a la siguiente:

```
try {
   // statements causing exception
} catch( ExceptionName e1 ) {
   // error handling code
} catch( ExceptionName e2 ) {
   // error handling code
} catch( ExceptionName eN ) {
   // error handling code
} finally {
   // statements to be executed
}
```

Puede enumerar varias declaraciones catch para detectar diferentes tipos de excepciones en caso de que su bloque try provoque más de una excepción en diferentes situaciones.

Clases de excepción en C

Las excepciones de C # están representadas por clases. Las clases de excepción en C # se derivan principalmente directa o indirectamente de la clase **System.Exception** . Algunas de las clases de excepción derivadas de la clase System.Exception son las clases **System.ApplicationException** y **System.SystemException** .

La clase **System.ApplicationException** admite excepciones generadas por programas de aplicación. Por lo tanto, las excepciones definidas por los programadores deben derivar de esta clase.

La clase **System.SystemException** es la clase base para todas las excepciones de sistema predefinidas.

La siguiente tabla proporciona algunas de las clases de excepción predefinidas derivadas de la clase Sytem.SystemException:

No Señor.	Clase de excepción y descripción
1	**System.IO.IOException** Maneja errores de E / S.
2	**System.IndexOutOfRangeException** Maneja los errores generados cuando un método hace referencia a un índice de matriz fuera de rango.
3	**System.ArrayTypeMismatchException** Maneja los errores generados cuando el tipo no coincide con el tipo de matriz.
4 4	**System.NullReferenceException** Maneja los errores generados al hacer referencia a un objeto nulo.
5 5	**System.DivideByZeroException** Maneja los errores generados al dividir un dividendo con cero.
6 6	**System.InvalidCastException** Maneja los errores generados durante la conversión de texto.
7 7	**System.OutOfMemoryException** Maneja errores generados por memoria libre insuficiente.
8	**System.StackOverflowException**

	Maneja los errores generados por el desbordamiento de la pila.

Manejo de excepciones

C # proporciona una solución estructurada para el manejo de excepciones en forma de bloques try and catch. Al usar estos bloques, las declaraciones del programa principal se separan de las declaraciones de manejo de errores.

Estos bloques de manejo de errores se implementan usando las palabras clave **try** , **catch** y **finalmente** . El siguiente es un ejemplo de lanzar una excepción cuando se produce una condición de división por cero:

```
using System;

namespace ErrorHandlingApplication {

   class DivNumbers {
      int result;

      DivNumbers() {
         result = 0;
      }

      public void division(int num1, int num2) {
         try {
            result = num1 / num2;
         } catch (DivideByZeroException e) {
            Console.WriteLine("Exception caught: {0}", e);
         } finally {
            Console.WriteLine("Result: {0}", result);
         }
      }

      static void Main(string[] args) {
         DivNumbers d = new DivNumbers();
         d.division(25, 0);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Exception caught: System.DivideByZeroException: Attempted to
divide by zero.
at ...
Result: 0
```

Crear excepciones definidas por el usuario

También puede definir su propia excepción. Las clases de excepción definidas por el usuario se derivan de la clase **Excepción** . El siguiente ejemplo demuestra esto:

```
using System;

namespace UserDefinedException {

   class TestTemperature {

      static void Main(string[] args) {
         Temperature temp = new Temperature();
         try {
            temp.showTemp();
         } catch(TempIsZeroException e) {
            Console.WriteLine("TempIsZeroException: {0}",
e.Message);
         }
         Console.ReadKey();
      }
   }
}

public class TempIsZeroException: Exception {

   public TempIsZeroException(string message): base(message)
{

   }
}

public class Temperature {
   int temperature = 0;

   public void showTemp() {

      if(temperature == 0) {
         throw (new TempIsZeroException("Zero Temperature
found"));
      } else {
         Console.WriteLine("Temperature: {0}", temperature);
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
TempIsZeroException: Zero Temperature found
```

Lanzar objetos

Puede lanzar un objeto si se deriva directa o indirectamente de la clase **System.Exception** . Puede usar una instrucción throw en el bloque catch para lanzar el objeto actual como:

```
Catch(Exception e) {
   ...
   Throw e
}
```

C # - E / S de archivo

Un **archivo** es una colección de datos almacenados en un disco con un nombre específico y una ruta de directorio. Cuando se abre un archivo para leer o escribir, se convierte en una **secuencia** .

La secuencia es básicamente la secuencia de bytes que pasan a través de la ruta de comunicación. Hay dos flujos principales: el **flujo de entrada** y el **flujo de salida** . El **flujo de entrada** se usa para leer datos del archivo (operación de lectura) y el **flujo de salida** se usa para escribir en el archivo (operación de escritura).

C # Clases de E / S

El espacio de nombres System.IO tiene varias clases que se utilizan para realizar numerosas operaciones con archivos, como crear y eliminar archivos, leer o escribir en un archivo, cerrar un archivo, etc.

La siguiente tabla muestra algunas clases no abstractas de uso común en el espacio de nombres System.IO:

No Señor.	Clase de E / S y descripción
1	**BinaryReader** Lee datos primitivos de una secuencia binaria.
2	**BinaryWriter** Escribe datos primitivos en formato binario.
3	**BufferedStream** Un almacenamiento temporal para una secuencia de bytes.
4 4	**Directorio** Ayuda a manipular una estructura de directorios.

5 5	**DirectoryInfo** Se utiliza para realizar operaciones en directorios.
6 6	**DriveInfo** Proporciona información para las unidades.
7 7	**Archivo** Ayuda a manipular archivos.
8	**Información del archivo** Se utiliza para realizar operaciones en archivos.
9 9	**FileStream** Se usa para leer y escribir en cualquier ubicación de un archivo.
10	**MemoryStream** Se utiliza para el acceso aleatorio a los datos transmitidos almacenados en la memoria.
11	**Camino** Realiza operaciones en la información de ruta.
12	**StreamReader** Se usa para leer caracteres de una secuencia de bytes.
13	**StreamWriter** Se utiliza para escribir caracteres en una secuencia.
14	**StringReader** Se utiliza para leer desde un búfer de cadena.
15	**StringWriter** Se usa para escribir en un búfer de cadena.

La clase FileStream

La clase **FileStream** en el espacio de nombres System.IO ayuda a leer, escribir y cerrar archivos. Esta clase deriva de la clase abstracta Stream.

Debe crear un objeto **FileStream** para crear un nuevo archivo o abrir un archivo existente. La sintaxis para crear un objeto **FileStream** es la siguiente:

```
FileStream <object_name> = new FileStream( <file_name>,
<FileMode Enumerator>,
   <FileAccess Enumerator>, <FileShare Enumerator>);
```

Por ejemplo, creamos un objeto FileStream **F** para leer un archivo llamado **sample.txt como se muestra** :

```
FileStream F = new FileStream("sample.txt", FileMode.Open,
FileAccess.Read,
   FileShare.Read);
```

No Señor.	Descripción de parámetros
1	**FileMode** El enumerador **FileMode** define varios métodos para abrir archivos. Los miembros del enumerador FileMode son: • **Anexar** : abre un archivo existente y coloca el cursor al final del archivo, o crea el archivo, si el archivo no existe. • **Crear** : crea un nuevo archivo. • **CreateNew** : especifica al sistema operativo que debe crear un nuevo archivo. • **Abrir** : abre un archivo existente. • **OpenOrCreate** : especifica al sistema operativo que debe abrir un archivo si existe, de lo contrario, debe crear un nuevo archivo. • **Truncar** : abre un archivo existente y trunca su tamaño a cero bytes.
2	**FileAccess** **FileAccess** encuestadores tienen miembros: **Leer** , **ReadWrite** y **escritura** .
3	**Recurso compartido de archivos** **Los** enumeradores de **FileShare** tienen los siguientes miembros: • **Heredable** : permite que un identificador de archivo pase la herencia a los procesos secundarios • **Ninguno** : rechaza compartir el archivo actual • **Leer** : permite abrir el archivo para leer. • **ReadWrite** : permite abrir el archivo para leer y escribir • **Escribir** : permite abrir el archivo para escribir

Ejemplo

El siguiente programa demuestra el uso de la clase **FileStream** :

```
using System;
using System.IO;

namespace FileIOApplication {

   class Program {

      static void Main(string[] args) {
         FileStream F = new FileStream("test.dat",
FileMode.OpenOrCreate,
            FileAccess.ReadWrite);

         for (int i = 1; i <= 20; i++) {
            F.WriteByte((byte)i);
         }

         F.Position = 0;
         for (int i = 0; i <= 20; i++) {
            Console.Write(F.ReadByte() + " ");
         }
         F.Close();
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

Operaciones avanzadas de archivos en C

El ejemplo anterior proporciona operaciones de archivo simples en C #. Sin embargo, para utilizar los inmensos poderes de las clases de C # System.IO, necesita conocer las propiedades y métodos de estas clases que se usan comúnmente.

No Señor.	Tema Descripción
1	Leer y escribir en archivos de texto Implica leer y escribir en archivos de texto. Las clases **StreamReader** y **StreamWriter** ayudan a lograrlo.
2	Leer y escribir en archivos binarios

	Implica leer y escribir en archivos binarios. Las clases **BinaryReader** y **BinaryWriter** ayudan a lograr esto.
3	Manipulando el sistema de archivos de Windows Le da a un programador de C # la capacidad de explorar y localizar archivos y directorios de Windows.

C # - Atributos

Un **atributo** es una etiqueta declarativa que se utiliza para transmitir información al tiempo de ejecución sobre los comportamientos de varios elementos como clases, métodos, estructuras, enumeradores, ensamblajes, etc. en su programa. Puede agregar información declarativa a un programa utilizando un atributo. Una etiqueta declarativa se representa entre corchetes ([]) colocados encima del elemento para el que se usa.

Los atributos se utilizan para agregar metadatos, como la instrucción del compilador y otra información como comentarios, descripción, métodos y clases a un programa. .Net Framework proporciona dos tipos de atributos: *los* atributos *predefinidos* y los atributos *personalizados* .

Especificar un atributo

La sintaxis para especificar un atributo es la siguiente:

```
[attribute(positional_parameters, name_parameter = value,
...)]
element
```

El nombre del atributo y sus valores se especifican entre corchetes, antes del elemento al que se aplica el atributo. Los parámetros posicionales especifican la información esencial y los parámetros de nombre especifican la información opcional.

Atributos predefinidos

.Net Framework proporciona tres atributos predefinidos:

- Atributo
- Condicional
- Obsoleto

Atributo

El atributo predefinido **AttributeUsage** describe cómo se puede utilizar una clase de atributo personalizado. Especifica los tipos de elementos a los que se puede aplicar el atributo.

La sintaxis para especificar este atributo es la siguiente:

```
[AttributeUsage (
   validon,
   AllowMultiple = allowmultiple,
   Inherited = inherited
)]
```

Dónde,

- El parámetro validon especifica los elementos del lenguaje en los que se puede colocar el atributo. Es una combinación del valor de un enumerador *AttributeTargets* . El valor predeterminado es *AttributeTargets.All* .
- El parámetro *allowmultiple* (opcional) proporciona el valor para la propiedad *AllowMultiple* de este atributo, un valor booleano. Si esto es cierto, el atributo es multiuso. El valor predeterminado es falso (uso único).
- El parámetro heredado (opcional) proporciona un valor para la propiedad *Heredada* de este atributo, un valor booleano. Si es cierto, el atributo es heredado por las clases derivadas. El valor predeterminado es falso (no heredado).

Por ejemplo,

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
```

Condicional

Este atributo predefinido marca un método condicional cuya ejecución depende de un identificador de preprocesamiento especificado.

Provoca la compilación condicional de llamadas a métodos, dependiendo del valor especificado como **Debug** o **Trace** . Por ejemplo, muestra los valores de las variables al depurar un código.

La sintaxis para especificar este atributo es la siguiente:

```
[Conditional(
   conditionalSymbol
)]
```

Por ejemplo,

```
[Conditional("DEBUG")]
```

El siguiente ejemplo demuestra el atributo:

```
#define DEBUG
using System;
using System.Diagnostics;

public class Myclass {
   [Conditional("DEBUG")]
```

```
   public static void Message(string msg) {
      Console.WriteLine(msg);
   }
}

class Test {
   static void function1() {
      Myclass.Message("In Function 1.");
      function2();
   }

   static void function2() {
      Myclass.Message("In Function 2.");
   }

   public static void Main() {
      Myclass.Message("In Main function.");
      function1();
      Console.ReadKey();
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
In Main function
In Function 1
In Function 2
```

Obsoleto

Este atributo predefinido marca una entidad de programa que no debe usarse. Le permite informar al compilador que descarte un elemento objetivo particular. Por ejemplo, cuando se usa un nuevo método en una clase y si aún desea conservar el método anterior en la clase, puede marcarlo como obsoleto mostrando un mensaje que debe usar el nuevo método, en lugar del método anterior.

La sintaxis para especificar este atributo es la siguiente:

```
[Obsolete (
   message
)]

[Obsolete (
   message,
   iserror
)]
```

Dónde,

- El *mensaje de* parámetro es una cadena que describe la razón por la cual el elemento está obsoleto y qué usar en su lugar.

- El parámetro *iserror* , es un valor booleano. Si su valor es verdadero, el compilador debe tratar el uso del elemento como un error. El valor predeterminado es falso (el compilador genera una advertencia).

El siguiente programa demuestra esto:

```
using System;

public class MyClass {
   [Obsolete("Don't use OldMethod, use NewMethod instead",
true)]

   static void OldMethod() {
      Console.WriteLine("It is the old method");
   }

   static void NewMethod() {
      Console.WriteLine("It is the new method");
   }

   public static void Main() {
      OldMethod();
   }
}
```

Cuando intenta compilar el programa, el compilador muestra un mensaje de error que indica:

```
Don't use OldMethod, use NewMethod instead
```

Crear atributos personalizados

.Net Framework permite la creación de atributos personalizados que se pueden usar para almacenar información declarativa y se pueden recuperar en tiempo de ejecución. Esta información puede estar relacionada con cualquier elemento objetivo, según los criterios de diseño y la necesidad de la aplicación.

Crear y usar atributos personalizados implica cuatro pasos:

- Declarando un atributo personalizado
- Construyendo el atributo personalizado
- Aplicar el atributo personalizado en un elemento del programa de destino
- Acceso a atributos a través de la reflexión

El último paso consiste en escribir un programa simple para leer los metadatos para encontrar varias anotaciones. Los metadatos son datos sobre datos o información utilizados para describir otros datos. Este programa debe usar reflexiones para acceder a los atributos en tiempo de ejecución. Esto lo discutiremos en el próximo capítulo.

Declarando un atributo personalizado

Un nuevo atributo personalizado debería derivarse de la clase **System.Attribute** . Por ejemplo,

```
//a custom attribute BugFix to be assigned to a class and its
members
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

En el código anterior, hemos declarado un atributo personalizado llamado *DeBugInfo* .

Construyendo el atributo personalizado

Construyamos un atributo personalizado llamado *DeBugInfo* , que almacena la información obtenida depurando cualquier programa. Deje que almacene la siguiente información:

- El número de código para el error
- Nombre del desarrollador que identificó el error.
- Fecha de la última revisión del código
- Un mensaje de cadena para almacenar los comentarios del desarrollador.

La clase *DeBugInfo* tiene tres propiedades privadas para almacenar las tres primeras informaciones y una propiedad pública para almacenar el mensaje. Por lo tanto, el número de error, el nombre del desarrollador y la fecha de revisión son los parámetros posicionales de la clase DeBugInfo y el mensaje es un parámetro opcional o con nombre.

Cada atributo debe tener al menos un constructor. Los parámetros posicionales deben pasarse a través del constructor. El siguiente código muestra la clase *DeBugInfo* :

```
//a custom attribute BugFix to be assigned to a class and its
members
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute {
   private int bugNo;
   private string developer;
   private string lastReview;
   public string message;
```

```
   public DeBugInfo(int bg, string dev, string d) {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
   }

   public int BugNo {
      get {
         return bugNo;
      }
   }

   public string Developer {
      get {
         return developer;
      }
   }

   public string LastReview {
      get {
         return lastReview;
      }
   }

   public string Message {
      get {
         return message;
      }
      set {
         message = value;
      }
   }
}
```

Aplicando el atributo personalizado

El atributo se aplica colocándolo inmediatamente antes de su objetivo:

```
[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return
type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused
variable")]
class Rectangle {
   //member variables
   protected double length;
   protected double width;
   public Rectangle(double l, double w) {
      length = l;
      width = w;
   }
   [DeBugInfo(55, "Zara Ali", "19/10/2012", Message = "Return
type mismatch")]
```

```
   public double GetArea() {
      return length * width;
   }
   [DeBugInfo(56, "Zara Ali", "19/10/2012")]

   public void Display() {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
   }
}
```

En el próximo capítulo, recuperamos información de atributos usando un objeto de clase Reflection.

C # - Reflexión

Los objetos de **reflexión** se utilizan para obtener información de tipo en tiempo de ejecución. Las clases que dan acceso a los metadatos de un programa en ejecución se encuentran en el **espacio de** nombres **System.Reflection** .

El **espacio de** nombres **System.Reflection** contiene clases que le permiten obtener información sobre la aplicación y agregar dinámicamente tipos, valores y objetos a la aplicación.

Aplicaciones de la reflexión

Reflection tiene las siguientes aplicaciones:

- Permite ver información de atributos en tiempo de ejecución.
- Permite examinar varios tipos en un ensamblaje e instanciar estos tipos.
- Permite la unión tardía a métodos y propiedades.
- Permite crear nuevos tipos en tiempo de ejecución y luego realiza algunas tareas utilizando esos tipos.

Visualización de metadatos

Hemos mencionado en el capítulo anterior que usando la reflexión puede ver la información del atributo.

El objeto **MemberInfo** de la clase **System.Reflection** debe inicializarse para descubrir los atributos asociados con una clase. Para hacer esto, define un objeto de la clase de destino, como -

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

El siguiente programa demuestra esto:

```
using System;
```

```
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute {
   public readonly string Url;

   public string Topic   // Topic is a named parameter {
      get {
         return topic;
      }

      set {
         topic = value;
      }
   }

   public HelpAttribute(string url)   // url is a positional
parameter {
      this.Url = url;
   }
   private string topic;
}

[HelpAttribute("Information on the class MyClass")]
class MyClass {

}

namespace AttributeAppl {

   class Program {

      static void Main(string[] args) {
         System.Reflection.MemberInfo info = typeof(MyClass);
         object[] attributes =
info.GetCustomAttributes(true);
         for (int i = 0; i < attributes.Length; i++) {
            System.Console.WriteLine(attributes[i]);
         }
         Console.ReadKey();
      }
   }
}
```

Cuando se compila y ejecuta, muestra el nombre de los atributos personalizados adjuntos a la clase *MyClass* :

```
HelpAttribute
```

Ejemplo

En este ejemplo, utilizamos el atributo *DeBugInfo* creado en el capítulo anterior y utilizamos la reflexión para leer metadatos en la clase *Rectangle* .

```
using System;
```

```
using System.Reflection;

namespace BugFixApplication {
   //a custom attribute BugFix to be
   //assigned to a class and its members
   [AttributeUsage(AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]

   public class DeBugInfo : System.Attribute {
      private int bugNo;
      private string developer;
      private string lastReview;
      public string message;

      public DeBugInfo(int bg, string dev, string d) {
         this.bugNo = bg;
         this.developer = dev;
         this.lastReview = d;
      }

      public int BugNo {
         get {
            return bugNo;
         }
      }

      public string Developer {
         get {
            return developer;
         }
      }

      public string LastReview {
         get {
            return lastReview;
         }
      }

      public string Message {
         get {
            return message;
         }
         set {
            message = value;
         }
      }
   }
   [DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return
type mismatch")]
```

```
   [DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused
variable")]

   class Rectangle {
      //member variables
      protected double length;
      protected double width;
      public Rectangle(double l, double w) {
         length = l;
         width = w;
      }

      [DeBugInfo(55, "Zara Ali", "19/10/2012", Message =
"Return type mismatch")]
      public double GetArea() {
         return length * width;
      }

      [DeBugInfo(56, "Zara Ali", "19/10/2012")]
      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle

   class ExecuteRectangle {
      static void Main(string[] args) {
         Rectangle r = new Rectangle(4.5, 7.5);
         r.Display();
         Type type = typeof(Rectangle);

         //iterating through the attribtues of the Rectangle
class
         foreach (Object attributes in
type.GetCustomAttributes(false)) {
            DeBugInfo dbi = (DeBugInfo)attributes;

            if (null != dbi) {
               Console.WriteLine("Bug no: {0}", dbi.BugNo);
               Console.WriteLine("Developer: {0}",
dbi.Developer);
               Console.WriteLine("Last Reviewed: {0}",
dbi.LastReview);
               Console.WriteLine("Remarks: {0}",
dbi.Message);
            }
         }

         //iterating through the method attribtues
         foreach (MethodInfo m in type.GetMethods()) {
```

```
            foreach (Attribute a in
m.GetCustomAttributes(true)) {
               DeBugInfo dbi = (DeBugInfo)a;

               if (null != dbi) {
                  Console.WriteLine("Bug no: {0}, for Method:
{1}", dbi.BugNo, m.Name);
                  Console.WriteLine("Developer: {0}",
dbi.Developer);
                  Console.WriteLine("Last Reviewed: {0}",
dbi.LastReview);
                  Console.WriteLine("Remarks: {0}",
dbi.Message);
               }
            }
         }
         Console.ReadLine();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Length: 4.5
Width: 7.5
Area: 33.75
Bug No: 49
Developer: Nuha Ali
Last Reviewed: 10/10/2012
Remarks: Unused variable
Bug No: 45
Developer: Zara Ali
Last Reviewed: 12/8/2012
Remarks: Return type mismatch
Bug No: 55, for Method: GetArea
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks: Return type mismatch
Bug No: 56, for Method: Display
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks:
```

C # - Propiedades

Las propiedades se denominan miembros de clases, estructuras e interfaces. Las variables o métodos miembros en una clase o estructuras se denominan **Campos** . Las propiedades son una extensión de campos y se accede utilizando la misma sintaxis. Utilizan **accesores a** través de los cuales los valores de los campos privados se pueden leer, escribir o manipular.

Las propiedades no nombran las ubicaciones de almacenamiento. En cambio, tienen **accesores** que leen, escriben o calculan sus valores.

Por ejemplo, tengamos una clase llamada Estudiante, con campos privados para edad, nombre y código. No podemos acceder directamente a estos campos desde fuera del alcance de la clase, pero podemos tener propiedades para acceder a estos campos privados.

Accesorios

El **descriptor** de **acceso** de una propiedad contiene las instrucciones ejecutables que ayudan a obtener (leer o computar) o configurar (escribir) la propiedad. Las declaraciones de descriptor de acceso pueden contener un descriptor de acceso get, un descriptor de acceso conjunto o ambos. Por ejemplo

```
// Declare a Code property of type string:
public string Code {
   get {
      return code;
   }
   set {
      code = value;
   }
}

// Declare a Name property of type string:
public string Name {
   get {
      return name;
   }
   set {
      name = value;
   }
}

// Declare a Age property of type int:
public int Age {
   get {
      return age;
   }
   set {
      age = value;
   }
}
```

Ejemplo

El siguiente ejemplo demuestra el uso de propiedades:

```
using System;
namespace postparaprogramadores {
   class Student {
      private string code = "N.A";
      private string name = "not known";
```

```
      private int age = 0;

      // Declare a Code property of type string:
      public string Code {
         get {
            return code;
         }
         set {
            code = value;
         }
      }

      // Declare a Name property of type string:
      public string Name {
         get {
            return name;
         }
         set {
            name = value;
         }
      }

      // Declare a Age property of type int:
      public int Age {
         get {
            return age;
         }
         set {
            age = value;
         }
      }
      public override string ToString() {
         return "Code = " + Code +", Name = " + Name + ", Age
= " + Age;
      }
   }

   class ExampleDemo {
      public static void Main() {

         // Create a new Student object:
         Student s = new Student();

         // Setting code, name and the age of the student
         s.Code = "001";
         s.Name = "Zara";
         s.Age = 9;
         Console.WriteLine("Student Info: {0}", s);

         //let us increase age
         s.Age += 1;
         Console.WriteLine("Student Info: {0}", s);
         Console.ReadKey();
```

```
        }
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

Propiedades abstractas

Una clase abstracta puede tener una propiedad abstracta, que debe implementarse en la clase derivada. El siguiente programa ilustra esto:

```
using System;
namespace postparaprogramadores {
   public abstract class Person {
      public abstract string Name {
         get;
         set;
      }
      public abstract int Age {
         get;
         set;
      }
   }

   class Student : Person {

      private string code = "N.A";
      private string name = "N.A";
      private int age = 0;

      // Declare a Code property of type string:
      public string Code {
         get {
            return code;
         }
         set {
            code = value;
         }
      }

      // Declare a Name property of type string:
      public override string Name {
         get {
            return name;
         }
         set {
            name = value;
         }
      }
```

```
      // Declare a Age property of type int:
      public override int Age {
         get {
            return age;
         }
         set {
            age = value;
         }
      }
      public override string ToString() {
         return "Code = " + Code +", Name = " + Name + ", Age
= " + Age;
      }
   }

   class ExampleDemo {
      public static void Main() {
         // Create a new Student object:
         Student s = new Student();

         // Setting code, name and the age of the student
         s.Code = "001";
         s.Name = "Zara";
         s.Age = 9;
         Console.WriteLine("Student Info:- {0}", s);

         //let us increase age
         s.Age += 1;
         Console.WriteLine("Student Info:- {0}", s);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

C # - Indizadores

Un **indexador** permite que un objeto se indexe, como una matriz. Cuando define un indexador para una clase, esta clase se comporta de manera similar a una **matriz virtual** . Luego puede acceder a la instancia de esta clase utilizando el operador de acceso a matriz ([]).

Sintaxis

Un indexador unidimensional tiene la siguiente sintaxis:

```
element-type this[int index] {

   // The get accessor.
   get {
```

```
      // return the value specified by index
   }

   // The set accessor.
   set {
      // set the value specified by index
   }
}
```

Uso de indexadores

La declaración de comportamiento de un indexador es, en cierta medida, similar a una propiedad. similar a las propiedades, usa **get** y **set** accessors para definir un indexador. Sin embargo, las propiedades devuelven o establecen un miembro de datos específico, mientras que los indexadores devuelven o establecen un valor particular de la instancia del objeto. En otras palabras, divide los datos de la instancia en partes más pequeñas e indexa cada parte, obtiene o establece cada parte.

Definir una propiedad implica proporcionar un nombre de propiedad. Los indexadores no se definen con nombres, sino con la palabra clave **this** , que se refiere a la instancia del objeto. El siguiente ejemplo demuestra el concepto:

```
using System;

namespace IndexerApplication {

   class IndexedNames {
      private string[] namelist = new string[size];
      static public int size = 10;

      public IndexedNames() {
         for (int i = 0; i < size; i++)
         namelist[i] = "N. A.";
      }

      public string this[int index] {

         get {
            string tmp;

            if( index >= 0 && index <= size-1 ) {
               tmp = namelist[index];
            } else {
               tmp = "";
            }

            return ( tmp );
         }
         set {
            if( index >= 0 && index <= size-1 ) {
               namelist[index] = value;
```

```
            }
         }
      }

      static void Main(string[] args) {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
         names[5] = "Sunil";
         names[6] = "Rubic";

         for ( int i = 0; i < IndexedNames.size; i++ ) {
            Console.WriteLine(names[i]);
         }
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
```

Indizadores sobrecargados

Los indexadores se pueden sobrecargar. Los indexadores también se pueden declarar con múltiples parámetros y cada parámetro puede ser de un tipo diferente. No es necesario que los índices tengan que ser enteros. C # permite que los índices sean de otros tipos, por ejemplo, una cadena.

El siguiente ejemplo muestra indexadores sobrecargados:

```
using System;

namespace IndexerApplication {

   class IndexedNames {
      private string[] namelist = new string[size];
      static public int size = 10;

      public IndexedNames() {
```

```
         for (int i = 0; i < size; i++) {
            namelist[i] = "N. A.";
         }
      }

      public string this[int index] {

         get {
            string tmp;

            if( index >= 0 && index <= size-1 ) {
               tmp = namelist[index];
            } else {
               tmp = "";
            }

            return ( tmp );
         }
         set {
            if( index >= 0 && index <= size-1 ) {
               namelist[index] = value;
            }
         }
      }

      public int this[string name] {

         get {
            int index = 0;

            while(index < size) {
               if (namelist[index] == name) {
                return index;
               }
               index++;
            }
            return index;
         }

      }

      static void Main(string[] args) {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
         names[5] = "Sunil";
         names[6] = "Rubic";

         //using the first indexer with int parameter
         for (int i = 0; i < IndexedNames.size; i++) {
```

```
            Console.WriteLine(names[i]);
         }

         //using the second indexer with the string parameter
         Console.WriteLine(names["Nuha"]);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
2
```

C # - Delegados

Los delegados de C # son similares a los punteros a las funciones, en C o C ++. Un **delegado** es una variable de tipo de referencia que contiene la referencia a un método. La referencia se puede cambiar en tiempo de ejecución.

Los delegados se utilizan especialmente para implementar eventos y los métodos de devolución de llamada. Todos los delegados se derivan implícitamente de la clase **System.Delegate** .

Declarando delegados

La declaración de delegado determina los métodos a los que puede hacer referencia el delegado. Un delegado puede referirse a un método, que tiene la misma firma que la del delegado.

Por ejemplo, considere un delegado:

```
public delegate int MyDelegate (string s);
```

El delegado anterior se puede usar para hacer referencia a cualquier método que tenga un único parámetro de *cadena* y devuelva una variable de tipo *int* .

La sintaxis para la declaración de delegado es:

```
delegate <return type> <delegate-name> <parameter list>
```

Delegados de instancia

Una vez que se declara un tipo de delegado, se debe crear un objeto delegado con la **nueva** palabra clave y asociarlo con un método en particular. Al crear un delegado, el argumento pasado a la **nueva** expresión se escribe de manera similar a una llamada al método, pero sin los argumentos del método. Por ejemplo

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

El siguiente ejemplo demuestra la declaración, la creación de instancias y el uso de un delegado que se puede utilizar para hacer referencia a métodos que toman un parámetro entero y devuelven un valor entero.

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl {

   class TestDelegate {
      static int num = 10;

      public static int AddNum(int p) {
         num += p;
         return num;
      }

      public static int MultNum(int q) {
         num *= q;
         return num;
      }

      public static int getNum() {
         return num;
      }

      static void Main(string[] args) {
         //create delegate instances
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);

         //calling the methods using the delegate objects
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());
         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Value of Num: 35
Value of Num: 175
```

Multidifusión de un delegado

Los objetos delegados se pueden componer utilizando el operador "+". Un delegado compuesto llama a los dos delegados de los que estaba compuesto. Solo se pueden componer delegados del mismo tipo. El operador "-" se puede utilizar para eliminar un delegado componente de un delegado compuesto.

Con esta propiedad de delegados, puede crear una lista de invocación de métodos a los que se llamará cuando se invoque a un delegado. Esto se llama **multidifusión** de un delegado. El siguiente programa demuestra la multidifusión de un delegado:

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl {

   class TestDelegate {
      static int num = 10;

      public static int AddNum(int p) {
         num += p;
         return num;
      }

      public static int MultNum(int q) {
         num *= q;
         return num;
      }

      public static int getNum() {
         return num;
      }

      static void Main(string[] args) {
         //create delegate instances
         NumberChanger nc;
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);
         nc = nc1;
         nc += nc2;

         //calling multicast
         nc(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
```

```
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Value of Num: 75
```

Usar delegados

El siguiente ejemplo demuestra el uso de delegado. El delegado *printString* se puede utilizar para hacer referencia al método que toma una cadena como entrada y no devuelve nada.

Usamos este delegado para llamar a dos métodos, el primero imprime la cadena en la consola y el segundo la imprime en un archivo:

```
using System;
using System.IO;

namespace DelegateAppl {

   class PrintString {
      static FileStream fs;
      static StreamWriter sw;

      // delegate declaration
      public delegate void printString(string s);

      // this method prints to the console
      public static void WriteToScreen(string str) {
         Console.WriteLine("The String is: {0}", str);
      }

      //this method prints to a file
      public static void WriteToFile(string s) {
         fs = new FileStream("c:\\message.txt",
         FileMode.Append, FileAccess.Write);
         sw = new StreamWriter(fs);
         sw.WriteLine(s);
         sw.Flush();
         sw.Close();
         fs.Close();
      }

      // this method takes the delegate as parameter and uses
it to
      // call the methods as required
      public static void sendString(printString ps) {
         ps("Hello World");
      }

      static void Main(string[] args) {
         printString ps1 = new printString(WriteToScreen);
         printString ps2 = new printString(WriteToFile);
```

```
            sendString(ps1);
            sendString(ps2);
            Console.ReadKey();
        }
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
The String is: Hello World
```

C # - Eventos

Los eventos son acciones del usuario, como presionar teclas, clics, movimientos del mouse, etc., o alguna ocurrencia, como notificaciones generadas por el sistema. Las aplicaciones deben responder a los eventos cuando ocurren. Por ejemplo, interrumpe. Los eventos se utilizan para la comunicación entre procesos.

Usar delegados con eventos

Los eventos se declaran y se generan en una clase y se asocian con los controladores de eventos utilizando delegados dentro de la misma clase o alguna otra clase. La clase que contiene el evento se usa para publicar el evento. Esto se llama la clase **publicador** . Alguna otra clase que acepta este evento se llama clase de **suscriptor** . Los eventos usan el modelo **editor-suscriptor** .

Un **editor** es un objeto que contiene la definición del evento y el delegado. La asociación delegado de eventos también se define en este objeto. Un objeto de clase publicador invoca el evento y se notifica a otros objetos.

Un **suscriptor** es un objeto que acepta el evento y proporciona un controlador de eventos. El delegado en la clase publicador invoca el método (controlador de eventos) de la clase suscriptor.

Declarando eventos

Para declarar un evento dentro de una clase, en primer lugar, debe declarar un tipo de delegado para el par como:

```
public delegate string BoilerLogHandler(string str);
```

luego, declare el evento usando la palabra clave del **evento** :

```
event BoilerLogHandler BoilerEventLog;
```

El código anterior define un delegado llamado *BoilerLogHandler* y un evento llamado *BoilerEventLog* , que invoca al delegado cuando se *genera* .

Ejemplo

```
using System;
```

```
namespace SampleApp {
   public delegate string MyDel(string str);

   class EventProgram {
      event MyDel MyEvent;

      public EventProgram() {
         this.MyEvent += new MyDel(this.WelcomeUser);
      }

      public string WelcomeUser(string username) {
         return "Welcome " + username;
      }

      static void Main(string[] args) {
         EventProgram obj1 = new EventProgram();
         string result =
obj1.MyEvent("Postparaprogramadores");
         Console.WriteLine(result);
      }

   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Welcome Postparaprogramadores
```

C # - Colecciones

Las clases de recopilación son clases especializadas para el almacenamiento y recuperación de datos. Estas clases proporcionan soporte para pilas, colas, listas y tablas hash. La mayoría de las clases de colección implementan las mismas interfaces.

Las clases de colección sirven para varios propósitos, como asignar memoria dinámicamente a elementos y acceder a una lista de elementos sobre la base de un índice, etc. Estas clases crean colecciones de objetos de la clase Object, que es la clase base para todos los tipos de datos en C #.

Diversas clases de colección y su uso

Las siguientes son las diversas clases comúnmente utilizadas del espacio de nombres **System.Collection** . Haga clic en los siguientes enlaces para verificar sus detalles.

No Señor.	Clase y descripción y uso
1	Lista de arreglo Representa la colección ordenada de un objeto que

	puede **indexarse** individualmente. Básicamente es una alternativa a una matriz. Sin embargo, a diferencia de la matriz, puede agregar y eliminar elementos de una lista en una posición específica utilizando un **índice** y la matriz cambia de tamaño automáticamente. También permite la asignación dinámica de memoria, agregar, buscar y ordenar elementos en la lista.
2	Tabla de picadillo Utiliza una **clave** para acceder a los elementos de la colección. Se utiliza una tabla hash cuando necesita acceder a elementos utilizando la clave, y puede identificar un valor clave útil. Cada elemento de la tabla hash tiene un par **clave / valor** . La clave se usa para acceder a los elementos de la colección.
3	SortedList Utiliza una **clave** y un **índice** para acceder a los elementos de una lista. Una lista ordenada es una combinación de una matriz y una tabla hash. Contiene una lista de elementos a los que se puede acceder utilizando una clave o un índice. Si accede a los elementos usando un índice, es una ArrayList, y si accede a los elementos usando una clave, es un Hashtable. La colección de artículos siempre se ordena por el valor clave.
4 4	Apilar Representa una colección de objetos de **último en entrar, primero en salir** . Se utiliza cuando necesita un acceso de elementos de último en entrar, primero en salir. Cuando agrega un elemento en la lista, se llama **empujar** el elemento y cuando lo elimina, se denomina **abrir** el elemento.
5 5	Cola Representa una colección de objetos **primero en** entrar **, primero en salir** . Se utiliza cuando necesita un acceso de los primeros en entrar, primero en salir. Cuando agrega un elemento en la lista, se llama en **cola** y cuando elimina un elemento, se llama **deque** .
6 6	BitArray Representa una matriz de la **representación binaria** utilizando los valores 1 y 0. Se utiliza cuando necesita almacenar los bits pero no conoce el número de bits por adelantado. Puede acceder a los elementos de la colección BitArray utilizando un **índice entero** , que comienza desde cero.

C # - Genéricos

Los genéricos le permiten definir la especificación del tipo de datos de los elementos de programación en una clase o un método, hasta que realmente se use en el programa. En otras palabras, los genéricos le permiten escribir una clase o método que puede funcionar con cualquier tipo de datos.

Usted escribe las especificaciones para la clase o el método, con parámetros sustitutos para los tipos de datos. Cuando el compilador encuentra un constructor para la clase o una llamada de función para el método, genera código para manejar el tipo de datos específico. Un ejemplo simple ayudaría a comprender el concepto:

```
using System;
using System.Collections.Generic;

namespace GenericApplication {

   public class MyGenericArray<T> {
      private T[] array;

      public MyGenericArray(int size) {
         array = new T[size + 1];
      }

      public T getItem(int index) {
         return array[index];
      }

      public void setItem(int index, T value) {
         array[index] = value;
      }
   }

   class Tester {
      static void Main(string[] args) {

         //declaring an int array
         MyGenericArray<int> intArray = new
MyGenericArray<int>(5);

         //setting values
         for (int c = 0; c < 5; c++) {
            intArray.setItem(c, c*5);
         }

         //retrieving the values
         for (int c = 0; c < 5; c++) {
            Console.Write(intArray.getItem(c) + " ");
         }

         Console.WriteLine();

         //declaring a character array
         MyGenericArray<char> charArray = new
MyGenericArray<char>(5);

         //setting values
         for (int c = 0; c < 5; c++) {
            charArray.setItem(c, (char)(c+97));
```

```
            }

            //retrieving the values
            for (int c = 0; c< 5; c++) {
               Console.Write(charArray.getItem(c) + " ");
            }
            Console.WriteLine();

            Console.ReadKey();
         }
      }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
0 5 10 15 20
a b c d e
```

Características de los genéricos

Generics es una técnica que enriquece sus programas de las siguientes maneras:

- Le ayuda a maximizar la reutilización del código, la seguridad de tipo y el rendimiento.
- Puede crear clases de colección genéricas. La biblioteca de clases de .NET Framework contiene varias nuevas clases de colecciones genéricas en el espacio de nombres *System.Collections.Generic* . Puede usar estas clases de colección genéricas en lugar de las clases de colección en el espacio de nombres *System.Collections* .
- Puede crear sus propias interfaces genéricas, clases, métodos, eventos y delegados.
- Puede crear clases genéricas restringidas para permitir el acceso a métodos en tipos de datos particulares.
- Puede obtener información sobre los tipos utilizados en un tipo de datos genérico en tiempo de ejecución mediante reflexión.

Métodos genéricos

En el ejemplo anterior, hemos usado una clase genérica; Podemos declarar un método genérico con un parámetro de tipo. El siguiente programa ilustra el concepto:

```
using System;
using System.Collections.Generic;

namespace GenericMethodAppl {

   class Program {

      static void Swap<T>(ref T lhs, ref T rhs) {
```

```
            T temp;
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }

        static void Main(string[] args) {
            int a, b;
            char c, d;
            a = 10;
            b = 20;
            c = 'I';
            d = 'V';

            //display values before swap:
            Console.WriteLine("Int values before calling
swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values before calling
swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            //call swap
            Swap<int>(ref a, ref b);
            Swap<char>(ref c, ref d);

            //display values after swap:
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling
swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            Console.ReadKey();
        }
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```

Delegados Genéricos

Puede definir un delegado genérico con parámetros de tipo. Por ejemplo

```
delegate T NumberChanger<T>(T n);
```

El siguiente ejemplo muestra el uso de este delegado:

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl {

   class TestDelegate {
      static int num = 10;

      public static int AddNum(int p) {
         num += p;
         return num;
      }

      public static int MultNum(int q) {
         num *= q;
         return num;
      }

      public static int getNum() {
         return num;
      }

      static void Main(string[] args) {
         //create delegate instances
         NumberChanger<int> nc1 = new
NumberChanger<int>(AddNum);
         NumberChanger<int> nc2 = new
NumberChanger<int>(MultNum);

         //calling the methods using the delegate objects
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());
         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Value of Num: 35
Value of Num: 175
```

C # - Métodos anónimos

Discutimos que los delegados se utilizan para hacer referencia a cualquier método que tenga la misma firma que la del delegado. En otras palabras, puede llamar a un método al que un delegado puede hacer referencia utilizando ese objeto delegado.

Los métodos anónimos proporcionan una técnica para pasar un bloque de código como parámetro delegado. Los métodos anónimos son los métodos sin nombre, solo el cuerpo.

No necesita especificar el tipo de retorno en un método anónimo; se infiere de la declaración de devolución dentro del cuerpo del método.

Escribir un método anónimo

Los métodos anónimos se declaran con la creación de la instancia delegada, con una palabra clave **delegada** . Por ejemplo,

```
delegate void NumberChanger(int n);
...
NumberChanger nc = delegate(int x) {
   Console.WriteLine("Anonymous Method: {0}", x);
};
```

El bloque de código *Console.WriteLine ("Método anónimo: {0}", x);* es el cuerpo del método anónimo.

El delegado podría llamarse tanto con métodos anónimos como con métodos nombrados de la misma manera, es decir, pasando los parámetros del método al objeto delegado.

Por ejemplo,

```
nc(10);
```

Ejemplo

El siguiente ejemplo demuestra el concepto:

```
using System;

delegate void NumberChanger(int n);
namespace DelegateAppl {

   class TestDelegate {
      static int num = 10;

      public static void AddNum(int p) {
         num += p;
         Console.WriteLine("Named Method: {0}", num);
      }

      public static void MultNum(int q) {
         num *= q;
         Console.WriteLine("Named Method: {0}", num);
      }

      public static int getNum() {
         return num;
```

```
      }

      static void Main(string[] args) {
         //create delegate instances using anonymous method
         NumberChanger nc = delegate(int x) {
            Console.WriteLine("Anonymous Method: {0}", x);
         };

         //calling the delegate using the anonymous method
         nc(10);

         //instantiating the delegate using the named methods
         nc =  new NumberChanger(AddNum);

         //calling the delegate using the named methods
         nc(5);

         //instantiating the delegate using another named
methods
         nc =  new NumberChanger(MultNum);

         //calling the delegate using the named methods
         nc(2);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Anonymous Method: 10
Named Method: 15
Named Method: 30
```

C # - Códigos inseguros

C # permite utilizar variables de puntero en una función de bloque de código cuando está marcado por el modificador **inseguro** . El **código inseguro** o el código no administrado es un bloque de código que utiliza una variable de **puntero** .

Nota - Para ejecutar los programas mencionados en este capítulo en codingground , configure la opción de compilación en *Proyecto >> Opciones de compilación >> Comando de compilación* en

```
mcs *.cs -out:main.exe -unsafe"
```

Punteros

Un **puntero** es una variable cuyo valor es la dirección de otra variable, es decir, la dirección directa de la ubicación de la memoria. similar a cualquier variable o constante, debe declarar un puntero antes de poder usarlo para almacenar cualquier dirección variable.

La forma general de una declaración de puntero es:

```
type *var-name;
```

Las siguientes son declaraciones válidas de puntero:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

El siguiente ejemplo ilustra el uso de punteros en C #, utilizando el modificador inseguro:

```
using System;

namespace UnsafeCodeApplication {

   class Program {

      static unsafe void Main(string[] args) {
         int var = 20;
         int* p = &var;
         Console.WriteLine("Data is: {0} ",  var);
         Console.WriteLine("Address is: {0}",  (int)p);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compiló y ejecutó, produce el siguiente resultado:

```
Data is: 20
Address is: 99215364
```

En lugar de declarar un método completo como inseguro, también puede declarar que una parte del código es insegura. El ejemplo en la siguiente sección muestra esto.

Recuperando el valor de los datos usando un puntero

Puede recuperar los datos almacenados en el lugar al que hace referencia la variable de puntero, utilizando el método **ToString ()** . El siguiente ejemplo demuestra esto:

```
using System;

namespace UnsafeCodeApplication {

   class Program {

      public static void Main() {

         unsafe {
            int var = 20;
            int* p = &var;
```

```
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p-
>ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }

         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compiló y ejecutó, produce el siguiente resultado:

```
Data is: 20
Data is: 20
Address is: 77128984
```

Pasar punteros como parámetros a métodos

Puede pasar una variable de puntero a un método como parámetro. El siguiente ejemplo ilustra esto:

```
using System;

namespace UnsafeCodeApplication {

   class TestPointer {

      public unsafe void swap(int* p, int *q) {
         int temp = *p;
         *p = *q;
         *q = temp;
      }

      public unsafe static void Main() {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2:
{1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}",
var1, var2);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

Acceso a elementos de matriz mediante un puntero

En C #, un nombre de matriz y un puntero a un tipo de datos igual que los datos de la matriz, no son el mismo tipo de variable. Por ejemplo, int * p e int [] p, no son del mismo tipo. Puede incrementar la variable de puntero p porque no está fija en la memoria pero una dirección de matriz está fija en la memoria, y no puede incrementar eso.

Por lo tanto, si necesita acceder a los datos de una matriz utilizando una variable de puntero, como lo hacemos tradicionalmente en C o C ++ (verifique: Punteros en C), debe corregir el puntero con la palabra clave **fija** .

El siguiente ejemplo demuestra esto:

```
using System;

namespace UnsafeCodeApplication {

   class TestPointer {

      public unsafe static void Main() {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* let us have array address in pointer */
         for ( int i = 0; i < 3; i++) {
            Console.WriteLine("Address of
list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i,
*(ptr + i));
         }

         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compiló y ejecutó, produce el siguiente resultado:

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

Compilación de código inseguro

Para compilar código inseguro, debe especificar el **modificador / inseguro** de línea de comandos con el compilador de línea de comandos.

Por ejemplo, para compilar un programa llamado prog1.cs que contiene código inseguro, desde la línea de comando, dé el comando:

```
csc /unsafe prog1.cs
```

Si está utilizando Visual Studio IDE, debe habilitar el uso de código inseguro en las propiedades del proyecto.

Para hacer esto

- Abra las **propiedades del proyecto** haciendo doble clic en el nodo de propiedades en el Explorador de soluciones.
- Haga clic en la pestaña **Construir** .
- Seleccione la opción " **Permitir código inseguro** ".

C # - Multithreading

Un **subproceso** se define como la ruta de ejecución de un programa. Cada hilo define un flujo de control único. Si su aplicación involucra operaciones complicadas y que requieren mucho tiempo, a menudo es útil establecer diferentes rutas de ejecución o subprocesos, con cada subproceso realizando un trabajo en particular.

Los hilos son **procesos ligeros** . Un ejemplo común de uso de hilo es la implementación de programación concurrente por parte de los sistemas operativos modernos. El uso de hilos ahorra el desperdicio del ciclo de la CPU y aumenta la eficiencia de una aplicación.

Hasta ahora, escribimos los programas donde se ejecuta un solo hilo como un solo proceso, que es la instancia en ejecución de la aplicación. Sin embargo, de esta manera la aplicación puede realizar un trabajo a la vez. Para que ejecute más de una tarea a la vez, podría dividirse en subprocesos más pequeños.

Ciclo de vida del hilo

El ciclo de vida de un subproceso comienza cuando se crea un objeto de la clase System.Threading.Thread y finaliza cuando el subproceso finaliza o completa la ejecución.

Los siguientes son los diversos estados en el ciclo de vida de un hilo:

- **El estado no iniciado** : es la situación en la que se crea la instancia del subproceso pero no se llama al método Start.
- **El estado Listo** : es la situación en la que el subproceso está listo para ejecutarse y esperando el ciclo de la CPU.
- **El estado** no ejecutable: un subproceso no es ejecutable cuando
    - El método del sueño ha sido llamado
    - El método de espera ha sido llamado
    - Bloqueado por operaciones de E / S
- **El estado muerto** : es la situación cuando el subproceso completa la ejecución o se aborta.

El hilo principal

En C #, la clase **System.Threading.Thread** se usa para trabajar con subprocesos. Permite crear y acceder a hilos individuales en una aplicación multiproceso. El primer hilo que se ejecutará en un proceso se llama hilo **principal** .

Cuando un programa C # inicia la ejecución, el hilo principal se crea automáticamente. Los subprocesos creados con la clase **Thread** se denominan subprocesos secundarios del subproceso principal. Puede acceder a un hilo utilizando la propiedad **CurrentThread** de la clase Thread.

El siguiente programa demuestra la ejecución del hilo principal:

```
using System;
using System.Threading;

namespace MultithreadingApplication {

   class MainThreadProgram {

      static void Main(string[] args) {
         Thread th = Thread.CurrentThread;
         th.Name = "MainThread";
         Console.WriteLine("This is {0}", th.Name);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
This is MainThread
```

Propiedades y métodos de la clase de subproceso

La siguiente tabla muestra algunas de las **propiedades** más utilizadas de la clase **Thread** :

No Señor.	Descripción de propiedad
1	**Contexto actual** Obtiene el contexto actual en el que se ejecuta el subproceso.
2	**Cultura actual** Obtiene o establece la cultura del subproceso actual.

3	**Principio actual** Obtiene o establece el principal actual del subproceso (para seguridad basada en roles).
4 4	**Tema actual** Obtiene el hilo actualmente en ejecución.
5 5	**CurrentUICulture** Obtiene o establece la cultura actual utilizada por el Administrador de recursos para buscar recursos específicos de la cultura en tiempo de ejecución.
6 6	**ExecutionContext** Obtiene un objeto ExecutionContext que contiene información sobre los diversos contextos del subproceso actual.
7 7	**IsAlive** Obtiene un valor que indica el estado de ejecución del subproceso actual.
8	**IsBackground** Obtiene o establece un valor que indica si un subproceso es o no un subproceso de fondo.
9 9	**IsThreadPoolThread** Obtiene un valor que indica si un subproceso pertenece o no al grupo de subprocesos administrados.
10	**ManagedThreadId** Obtiene un identificador único para el subproceso administrado actual.
11	**Nombre** Obtiene o establece el nombre del subproceso.
12	**Prioridad** Obtiene o establece un valor que indica la prioridad de programación de un subproceso.
13	**ThreadState**

	Obtiene un valor que contiene los estados del subproceso actual.

La siguiente tabla muestra algunos de los **métodos** más utilizados de la clase **Thread** :

No Señor.	Método y descripción
1	**anulación pública Abortar ()** Genera una ThreadAbortException en el hilo en el que se invoca, para comenzar el proceso de terminar el hilo. Llamar a este método generalmente termina el hilo.
2	**público estático LocalDataStoreSlot AllocateDataSlot ()** Asigna una ranura de datos sin nombre en todos los hilos. Para un mejor rendimiento, utilice los campos que están marcados con el atributo ThreadStaticAttribute en su lugar.
3	**público estático LocalDataStoreSlot AllocateNamedDataSlot (nombre de cadena)** Asigna una ranura de datos con nombre en todos los hilos. Para un mejor rendimiento, utilice los campos que están marcados con el atributo ThreadStaticAttribute en su lugar.
4 4	**public static void BeginCriticalRegion ()** Notifica a un host que la ejecución está a punto de ingresar una región de código en la que los efectos de una anulación de subproceso o una excepción no controlada pueden poner en peligro otras tareas en el dominio de la aplicación.
5 5	**public static void BeginThreadAffinity ()** Notifica a un host que el código administrado está a punto de ejecutar instrucciones que dependen de la identidad del subproceso físico actual del sistema operativo.
6 6	**público estático vacío EndCriticalRegion ()** Notifica a un host que la ejecución está a punto de ingresar una región de código en la que los efectos de una anulación de subproceso o una excepción no controlada se limitan a la tarea actual.
7 7	**público estático vacío EndThreadAffinity ()** Notifica a un host que el código administrado ha terminado de ejecutar instrucciones que dependen de la identidad del subproceso físico actual del sistema operativo.

8	**public static void FreeNamedDataSlot (nombre de cadena)** Elimina la asociación entre un nombre y un espacio, para todos los hilos en el proceso. Para un mejor rendimiento, utilice los campos que están marcados con el atributo ThreadStaticAttribute en su lugar.
9 9	**GetData de objeto estático público (ranura LocalDataStoreSlot)** Recupera el valor de la ranura especificada en el hilo actual, dentro del dominio actual del hilo actual. Para un mejor rendimiento, utilice los campos que están marcados con el atributo ThreadStaticAttribute en su lugar.
10	**AppDomain público estático GetDomain ()** Devuelve el dominio actual en el que se ejecuta el hilo actual.
11	**AppDomain público estático GetDomainID ()** Devuelve un identificador de dominio de aplicación único
12	**LocalDataStoreSlot público estático GetNamedDataSlot (nombre de cadena)** Busca una ranura de datos con nombre. Para un mejor rendimiento, utilice los campos que están marcados con el atributo ThreadStaticAttribute en su lugar.
13	**interrupción pública nula ()** Interrumpe un hilo que está en el estado de hilo WaitSleepJoin.
14	**public void Join ()** Bloquea el subproceso de llamada hasta que termina un subproceso, mientras continúa realizando el bombeo estándar de COM y SendMessage. Este método tiene diferentes formularios sobrecargados.
15	**público estático vacío MemoryBarrier ()** Sincroniza el acceso a la memoria de la siguiente manera: el procesador que ejecuta el subproceso actual no puede reordenar las instrucciones de tal manera que los accesos a la memoria antes de la llamada a MemoryBarrier se ejecuten después de los accesos a la memoria que siguen a la llamada a MemoryBarrier.
dieciséis	**public static void ResetAbort ()** Cancela una cancelación solicitada para el hilo actual.
17	**SetData público estático vacío (ranura LocalDataStoreSlot, datos de objeto)** Establece los datos en la ranura especificada en el hilo actualmente en ejecución, para el dominio actual de ese hilo. Para un mejor rendimiento, utilice los campos

	marcados con el atributo ThreadStaticAttribute en su lugar.
18 años	**inicio público vacío ()** Inicia un hilo.
19	**sueño estático público vacío (int millisecondsTimeout)** Hace que el hilo se detenga por un período de tiempo.
20	**público estático vacío SpinWait (int iteraciones)** Hace que un hilo espere el número de veces definido por el parámetro iteraciones
21	**Byte estático público VolatileRead (dirección de byte de referencia)** **public static double VolatileRead (dirección doble de referencia)** **public static int VolatileRead (ref int address)** **Public static Object VolatileRead (ref. Dirección del objeto)** Lee el valor de un campo. El valor es el último escrito por cualquier procesador en una computadora, independientemente de la cantidad de procesadores o el estado de la memoria caché del procesador. Este método tiene diferentes formularios sobrecargados. Solo algunos se dan arriba.
22	**público estático vacío VolatileWrite (dirección de byte de referencia, valor de byte)** **público estático vacío VolatileWrite (dirección doble de referencia, valor doble)** **público estático void VolatileWrite (ref int address, int value)** **público estático vacío VolatileWrite (ref Dirección de objeto, valor de objeto)** Escribe un valor en un campo inmediatamente, de modo que el valor sea visible para todos los procesadores de la computadora. Este método tiene diferentes formularios sobrecargados. Solo algunos se dan arriba.
23	**rendimiento estático público bool ()** Hace que el hilo de llamada produzca ejecución en otro hilo que está listo para ejecutarse en el procesador actual. El sistema operativo selecciona el hilo a ceder.

Crear hilos

Los hilos se crean extendiendo la clase de hilos. La clase Thread extendida luego llama al método **Start ()** para comenzar la ejecución del hilo secundario.

El siguiente programa demuestra el concepto:

```
using System;
using System.Threading;

namespace MultithreadingApplication {

   class ThreadCreationProgram {

      public static void CallToChildThread() {
         Console.WriteLine("Child thread starts");
      }

      static void Main(string[] args) {
         ThreadStart childref = new
ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child
thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
In Main: Creating the Child thread
Child thread starts
```

Administrar hilos

La clase Thread proporciona varios métodos para administrar hilos.

El siguiente ejemplo demuestra el uso del método **sleep ()** para hacer una pausa de hilo por un período específico de tiempo.

```
using System;
using System.Threading;

namespace MultithreadingApplication {

   class ThreadCreationProgram {

      public static void CallToChildThread() {
         Console.WriteLine("Child thread starts");

         // the thread is paused for 5000 milliseconds
         int sleepfor = 5000;

         Console.WriteLine("Child Thread Paused for {0}
seconds", sleepfor / 1000);
         Thread.Sleep(sleepfor);
         Console.WriteLine("Child thread resumes");
      }
```

```
      static void Main(string[] args) {
         ThreadStart childref = new
ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child
thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

Destruyendo hilos

El método **Abort ()** se usa para destruir hilos.

El tiempo de ejecución anula el subproceso lanzando una **ThreadAbortException** . Esta excepción no se puede detectar, el control se envía al bloque *finalmente* , si lo hay.

El siguiente programa ilustra esto:

```
using System;
using System.Threading;

namespace MultithreadingApplication {

   class ThreadCreationProgram {

      public static void CallToChildThread() {

         try {
            Console.WriteLine("Child thread starts");

            // do some work, like counting to 10
            for (int counter = 0; counter <= 10; counter++) {
               Thread.Sleep(500);
               Console.WriteLine(counter);
            }

            Console.WriteLine("Child Thread Completed");
         } catch (ThreadAbortException e) {
            Console.WriteLine("Thread Abort Exception");
         } finally {
            Console.WriteLine("Couldn't catch the Thread
Exception");
         }
```

```
      }

      static void Main(string[] args) {
         ThreadStart childref = new
ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child
thread");
         Thread childThread = new Thread(childref);
         childThread.Start();

         //stop the main thread for some time
         Thread.Sleep(2000);

         //now abort the child
         Console.WriteLine("In Main: Aborting the Child
thread");

         childThread.Abort();
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
In Main: Creating the Child thread
Child thread starts
0
1
2
In Main: Aborting the Child thread
Thread Abort Exception
Couldn't catch the Thread Exception
```