PROGRAMACIÓN MULTIMEDIA

Y

DISPOSITIVOS MÓVILES

UT1. Introducción al lenguaje C#

Departamento de Informática y Comunicaciones

CFGS Desarrollo de Aplicaciones Multiplataforma

Segundo Curso

IES Virrey Morcillo

Villarrobledo (Albacete)

Índice

1. El lenguaje de programación C# 3
2. Estructura típica de un programa 4
3. Léxico del lenguaje C# 6
   3.1. Comentarios 6
   3.2. Identificadores 6
   3.3. Palabras reservadas 7
   3.4. Expresiones y operadores 7
      3.4.1. Operadores aritméticos 8
      3.4.2. Operadores de comparación 9
      3.4.3. Operadores lógicos 9
      3.4.4. Operador de asignación 9
4. Tipos de datos y variables 10
5. Sentencias o instrucciones 12
   5.1. Sentencia de asignación 13
   5.2. Sentencias de selección 14
   5.3. Sentencias de iteración 15
6. Enumeraciones 17
7. Estructuras 18
8. Matrices 19
9. Colecciones 21

1. El lenguaje de programación C#

C# es un lenguaje de programación simple, moderno, orientado a objetos y fuertemente tipado. Sus raíces en la familia de lenguajes C hacen que C# sea familiar para los programadores C, C ++, Java y JavaScript.

La **sintaxis de C#** es muy expresiva, pero también sencilla y fácil de aprender. Cualquier persona familiarizada con C, C++ o Java, reconocerá al instante la sintaxis de llaves de C#.

En cuanto **lenguaje orientado a objetos**, C# admite los conceptos de encapsulación, herencia y polimorfismo. Todas las variables y métodos, incluido el método Main, el punto de entrada de la aplicación, se encapsulan dentro de las definiciones de clase.

El lenguaje C# también incluye compatibilidad para **programación orientada a componentes**. El diseño de software contemporáneo se basa cada vez más en componentes de software en forma de paquetes independientes y auto-descriptivos de funcionalidad. La clave de estos componentes es que presentan un modelo de programación con propiedades, métodos y eventos; tienen atributos que proporcionan información declarativa sobre el componente; e incorporan su propia documentación. C# proporciona construcciones de lenguaje para admitir directamente estos conceptos, por lo que se trata de un lenguaje muy natural en el que crear y usar componentes de software.

Varias características de C# ayudan en la construcción de aplicaciones sólidas y duraderas: la **recolección de elementos no utilizados** automáticamente libera la memoria ocupada por objetos no utilizados y no accesibles; el **control de excepciones** proporciona un enfoque estructurado y extensible para la detección de errores y la recuperación; y el diseño del lenguaje con **seguridad de tipos** hace imposible leer desde variables sin inicializar, indexar matrices más allá de sus límites o realizar conversiones de tipos no comprobados.

C# tiene un **sistema de tipo unificado**. Todos los tipos de C#, incluidos los tipos primitivos como int y double, se heredan de un único tipo raíz llamado object. Por lo tanto, todos los tipos comparten un conjunto de operaciones comunes, y los valores de todos los tipos se pueden almacenar, transportar y utilizar de manera coherente. Además, C# admite **tipos de valor** y **tipos de referencia** definidos por el usuario, lo que permite la asignación dinámica de objetos, así como almacenamiento en línea de estructuras ligeras.

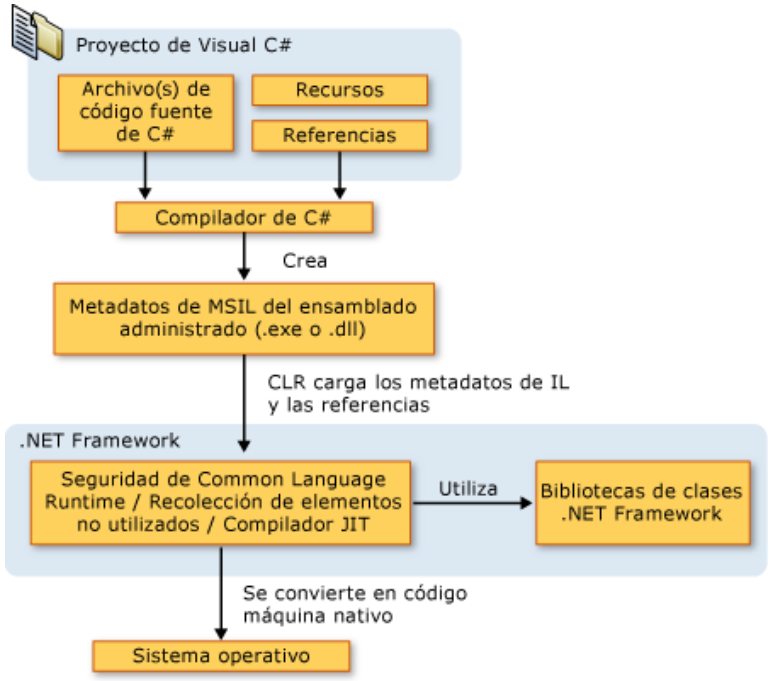
Los **programas de C# se ejecutan en .NET**, un componente integral de Windows que incluye un sistema de ejecución virtual llamado Common Language Runtime (CLR) y un conjunto unificado de bibliotecas de clases (BCL). El CLR es la implementación comercial de Microsoft de Common Language Infrastructure (CLI), un estándar internacional que es la base para la creación de entornos de ejecución y desarrollo en los que los lenguajes y las bibliotecas trabajan juntos sin problemas.

El código fuente escrito en C# se **compila** en un lenguaje intermedio (IL) que guarda conformidad con la especificación de CLI. El código y los recursos IL, como mapas de bits y cadenas, se almacenan en disco en un archivo ejecutable denominado **ensamblado**, normalmente con la extensión .exe o .dll. Un ensamblado contiene un manifiesto que proporciona información sobre los tipos, la versión, la referencia cultural y los requisitos de seguridad del ensamblado.

Cuando se ejecuta el programa de C#, el ensamblado se carga en el CLR, el cual podría realizar diversas acciones en función de la información en el manifiesto. Si se cumplen los requisitos de seguridad, el CLR realiza la compilación Just in time (JIT) para convertir el código IL en instrucciones máquina nativas.

El CLR también proporciona otros servicios relacionados con la recolección de elementos no utilizados, el control de excepciones y la administración de recursos. El código que se ejecuta en el CLR se conoce a veces como "código administrado", a diferencia del "código no administrado" que se compila en lenguaje de máquina nativo destinado a un sistema específico.

En la siguiente figura se ilustran las relaciones de tiempo de compilación y tiempo de ejecución de archivos de código fuente de C#, las bibliotecas de clases de .NET Framework, los ensamblados y el CLR.



La **interoperabilidad entre lenguajes** es una característica principal de .NET Framework. Debido a que el código IL generado por el compilador de C# cumple la especificación de tipo común (CTS), este código puede interactuar con el código generado a partir de las versiones .NET de Visual Basic, Visual C++ o cualquiera de los más de 20 lenguajes compatibles con CTS. Un solo ensamblado puede contener varios módulos escritos en diferentes lenguajes .NET y los tipos se pueden hacer referencia mutuamente como si estuvieran escritos en el mismo lenguaje.

2. Estructura típica de un programa

Los principales conceptos organizativos en C# son **programas**, **espacios de nombres**, **tipos**, **miembros** y **ensamblados**.

Un **programa** en C# es una secuencia de caracteres que el compilador se encarga de agrupar convenientemente para reconocer las diferentes sentencias del lenguaje, rechazando las construcciones que no sean válidas.

Los programas de C# constan de uno o más archivos de origen. Los programas declaran tipos, que contienen miembros y pueden organizarse en espacios de nombres. Las clases e interfaces son ejemplos de tipos. Los campos, los métodos, las propiedades y los eventos son ejemplos de miembros.

Cuando se compilan programas de C#, se empaquetan físicamente en ensamblados. Normalmente, los ensamblados tienen la extensión de archivo .exe o .dll, dependiendo de si implementan aplicaciones o bibliotecas, respectivamente.

En un primer ejemplo se presenta el típico programa "Hola mundo":

```
using System;
namespace Hola {
   class Program {
          static void Main() {
                 Console.WriteLine("Hola mundo!");
                 Console.ReadKey();
          }
   }
}
```

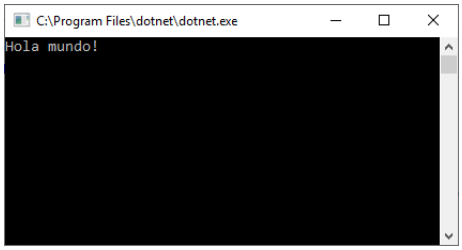
El programa "Hola mundo" empieza con una directiva using que hace referencia al espacio de nombres System. Los espacios de nombres proporcionan un método jerárquico para organizar las bibliotecas y los programas de C#. Los espacios de nombres contienen tipos y otros espacios de nombres; por ejemplo, el espacio de nombres System contiene varios tipos, como la clase Console a la que se hace referencia en el programa, y otros espacios de nombres, como IO y Collections. Una directiva using que hace referencia a un espacio de nombres determinado permite el uso no calificado de los tipos que son miembros de ese espacio de nombres. Debido a la directiva using, puede utilizar el programa Console.WriteLine como abreviatura de System.Console.WriteLine.

La palabra clave namespace se usa para declarar el ámbito llamado Hola que contiene un conjunto de objetos relacionados. Podemos usar un espacio de nombres para organizar los elementos de código y crear tipos únicos globales.

La clase Program declarada por el programa "Hola mundo" tiene un miembro único, el método llamado Main. El método Main se declara con el modificador static. Por convención, un método estático denominado Main sirve como punto de entrada de un programa.

La salida del programa la genera el método WriteLine y ReadKey de la clase Console en el espacio de nombres System. Esta clase la proporcionan las bibliotecas de clase estándar, a las que, de forma predeterminada, el compilador hace referencia automáticamente.

Normalmente, los archivos de código fuente de C# tienen la extensión de archivo .cs. Suponiendo que el programa "Hola mundo" se almacena en el archivo hola.cs. El programa podría compilarse y generaría un ensamblado ejecutable denominado hola.exe. La salida que genera la aplicación cuando se ejecuta es:



Para crear una solución o proyecto nuevo en Visual Studio debemos seguir los siguientes pasos:

1. Inicia Visual Studio Community.
2. En la barra de menús, elije Archivo, Nuevo, Proyecto.
3. Aparece el cuadro de diálogo Nuevo proyecto.
4. Expande Instalado, Plantillas, Visual C#, .NET Core y, luego, elije Aplicación de consola (.NET Core).

5. En el cuadro Nombre, escribe un nombre para el proyecto y, después, clic el botón Aceptar.
6. El proyecto nuevo aparece en el Explorador de soluciones.
7. Si Program.cs no está abierto en el Editor de código, abre el menú contextual de Program.cs en el Explorador de soluciones y elije Ver código.
8. Reemplaza el contenido de Program.cs por el código anterior.
9. Pulsa la tecla F5 para ejecutar el proyecto y aparecerá una ventana del símbolo del sistema.

3. Léxico del lenguaje C#

El léxico del lenguaje está formado por el conjunto de símbolos que se pueden usar en el lenguaje. Estos símbolos o elementos básicos del lenguaje, podrán ser de los siguientes: comentarios, identificadores, palabras reservadas, expresiones y operadores.

3.1. Comentarios

Un **comentario** es texto que incluido en el código fuente de un programa con la idea de facilitar su legibilidad a los programadores y cuyo contenido es, por defecto, completamente ignorado por el compilador.

Los caracteres // convierten el resto de la línea en un comentario. También se puede convertir un bloque de texto en un comentario escribiéndolo entre los caracteres /* y */.

3.2. Identificadores

Un **identificador** es el nombre que se asigna a un tipo (clase, interfaz, struct, delegado o enumeración), miembro, variable o espacio de nombres. Típicamente el nombre de un identificador será una secuencia de cualquier número de caracteres alfanuméricos. Los identificadores válidos deben seguir estas reglas:

- ✓ Los identificadores deben comenzar con una letra, o _.
- ✓ Los identificadores pueden contener caracteres de letra Unicode, caracteres de dígito decimales, caracteres de conexión Unicode, caracteres de combinación Unicode o caracteres de formato Unicode.

Además de las reglas, hay una serie de convenciones de nomenclatura de los identificadores que se usa en las API de .NET. Por convención, los programas de C# usan PascalCase para nombres de tipo, espacios de nombres y todos los miembros públicos. Además, son comunes las convenciones siguientes:

- ✓ Los nombres de interfaz empiezan por una I mayúscula.
- ✓ Los tipos de atributo terminan con la palabra Attribute.
- ✓ Los tipos de enumeración usan un sustantivo singular para los que no son marcas y uno plural para los que sí.
- ✓ Los identificadores no deberían contener dos caracteres _ consecutivos. Esos nombres están reservados para los identificadores generados por el compilador.

3.3. Palabras reservadas

Ciertos identificadores tienen un significado predeterminado en el lenguaje y sólo pueden usarse en el contexto en el que han sido definidas. Por este motivo no pueden crearse objetos utilizando estos nombres reservados.

A estos identificadores reservados predefinidos que tienen un significado especial para el compilador se les conoce como **palabras reservadas o palabras clave**. Éstas no se podrán utilizar como identificadores en el programa a no ser que incluyan @ como prefijo. Por ejemplo, @if es un identificador válido, pero if no lo es, porque if es una palabra reservadas.

A continuación, se muestran las palabras clave que son identificadores reservados en cualquier parte de un programa en C#:

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

3.4. Expresiones y operadores

Una **expresión** es una secuencia de uno o más operandos y cero o más operadores que se pueden evaluar como un valor, objeto, método o espacio de nombres único. Las expresiones pueden constar de un valor literal, una invocación de método, un operador y sus operandos o un nombre simple. Los nombres simples pueden ser el nombre de una variable, el miembro de un tipo, el parámetro de un método, un espacio de nombres o un tipo.

Todas las expresiones son evaluadas a un simple valor cuando la aplicación se ejecuta. El tipo de valor que una expresión genera depende de los tipos de operandos y operadores que utilicemos.

Un **operador** es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado.

Los operadores que toman un operando, como el operador de incremento (++) o new, se conocen como operadores unarios. Los operadores que toman dos operandos, como los operadores aritméticos (+ - * /) se conocen como operadores binarios. El operador condicional (? :) consodera tres operandos.

En la siguiente tabla se muestran los distintos tipos de operadores:

Tipo	Operadores
Aritméticos	+, -, *, /, %
Incremento, decremento	++, --
Comparación	==, !=, <, >, <=, >=, is
Concatenación de cadenas	+
Operaciones lógicas de bits	&, |, ^, !, ~, &&, ||
Indizado (el contador inicia en el elemento 0)	[ ]
Conversiones	( ), as
Asignación	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, ??
Rotación de Bits	<<, >>
Información de Tipos de datos	sizeof, typeof
Concatenación y eliminación de Delegados	+, -
Control de excepción de Overflow	checked, unchecked
Apuntadores y direccionamiento en código No seguro (Unsafe code)	*, ->, [ ], &
Condicional (operador ternario)	?:

En una expresión que contiene varios operadores, el orden de aplicación de estos viene determinado por la prioridad del operador, la asociatividad y los paréntesis. Por ejemplo, la expresión x + y * z se evalúa como x + (y * z) porque el operador * tiene mayor precedencia que el operador +.

Cuando un operando se encuentra entre dos operadores con la misma precedencia, la asociatividad de los operadores controla el orden en que se realizan las operaciones:

- ✓ Excepto los operadores de asignación, todos los operadores binarios son asociativos a la izquierda, lo que significa que las operaciones se realizan de izquierda a derecha. Por ejemplo, x + y + z se evalúa como (x + y) + z.
- ✓ Los operadores de asignación y el operador condicional (? :) son asociativos a la derecha, lo que significa que las operaciones se realizan de derecha a izquierda. Por ejemplo, x = y = z se evalúa como x = (y = z).

La precedencia y la asociatividad pueden controlarse mediante paréntesis.

3.4.1. Operadores aritméticos

Los **operadores aritméticos** incluidos son los típicos de:

- ✓ suma (+),
- ✓ resta (-),
- ✓ producto (*),
- ✓ división (/),
- ✓ módulo (%)
- ✓ también se incluyen operadores de menos unario (–) y más unario (+).

Relacionados con las operaciones aritméticas se encuentran un par de operadores llamados checked y unchecked que permiten controlar si se desea detectar los desbordamientos que puedan

producirse si al realizar este tipo de operaciones el resultado es superior a la capacidad del tipo de datos de sus operandos.

3.4.2. Operadores de comparación

Los **operadores de comparación** se han incluido los tradicionales operadores de:

- ✓ igualdad (==),
- ✓ desigualdad (!=),
- ✓ mayor que (>),
- ✓ menor que (<),
- ✓ mayor o igual que (>=) y
- ✓ menor o igual que (<=).

3.4.3. Operadores lógicos

Los **operadores lógicos** permiten realizar las operaciones lógicas típicas:

- ✓ and (&& y &),
- ✓ or (|| y |),
- ✓ not (!) y
- ✓ xor (^).

Los operadores && y || se diferencia de & y | en que los primeros realizan evaluación en cortocircuito y los segundos no. La evaluación en cortocircuito consiste en lo siguiente: si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente, mientras que la evaluación normal consiste en evaluar siempre ambos operandos.

3.4.4. Operador de asignación

Para realizar asignaciones se usa el **operador =**, operador que además de realizar la asignación que se le solicita devuelve el valor asignado. Por ejemplo, la expresión a = b asigna a la variable a el valor de la variable b y devuelve dicho valor, mientras que la expresión c = a = b asigna a c y a el valor de b (el operador = es asociativo por la derecha).

También se han incluido operadores de asignación compuestos que permiten ahorrar escritura de código a la hora de realizar asignaciones tan comunes como:

```
contador = contador + 100;
contador += 100;
```

Las dos líneas anteriores son equivalentes, pues el operador compuesto += lo que hace es asignar a su primer operando el valor que tenía más el valor de su segundo operando.

Aparte del operador de asignación compuesto +=, también se ofrecen operadores de asignación compuestos para la mayoría de los operadores binarios ya vistos. Estos son: +=, -=, *=, /=, %=, &=, |=, ^=, <<= y >>=.

Otros dos operadores de asignación incluidos son los de incremento (++) y decremento (--) Estos operadores permiten, respectivamente, aumentar y disminuir en una unidad el valor de la variable sobre el que se aplican.

4. Tipos de datos y variables

C# es un lenguaje fuertemente tipado. Todas las variables y constantes tienen un tipo, al igual que todas las expresiones que se evalúan como un valor, cada una de las firmas o prototipos de un método especifica un tipo para cada parámetro de entrada y para el valor devuelto. De esta manera, los **tipos de datos** de una variable determinan el conjunto de valores que pueden tomar y las operaciones que pueden realizarse con ella.

La biblioteca de clases .NET Framework define un conjunto de tipos numéricos integrados, así como tipos más complejos que representan una amplia variedad de construcciones lógicas, como el sistema de archivos, conexiones de red, colecciones y matrices de objetos, y fechas.

Hay dos **clases de tipos** en C#:

- ✓ tipos de valor y
- ✓ tipos de referencia.

Las variables de **tipos de valor** contienen directamente los datos, mientras que las variables de los **tipos de referencia** almacenan referencias a los datos, lo que se conoce como objetos. Con los tipos de referencia, es posible que dos variables hagan referencia al mismo objeto y que, por tanto, las operaciones en una variable afecten al objeto al que hace referencia la otra variable. Con los tipos de valor, cada variable tiene su propia copia de los datos y no es posible que las operaciones en una variable afecten a la otra.

Los **tipos de valor de C#** se dividen en:

- tipos simples,
- tipos de enumeración,
- tipos de estructura y
- tipos de valores NULL.

Los **tipos de referencia de C#** se dividen en:

- tipos de clase,
- tipos de interfaz,
- tipos de matriz y
- tipos delegados.

A continuación se proporciona información general del sistema de tipos de C#:

1. Tipos de valor
    a. Tipos simples
        i. Entero con signo: sbyte, short, int, long
        ii. Entero sin signo: byte, ushort, uint, ulong
        iii. Caracteres Unicode: char

iv. Punto flotante de IEEE: float, double
v. Decimal de alta precisión: decimal
vi. Booleano: bool

b. Tipos de enumeración
    i. Tipos definidos por el usuario con el formato enum E {...}
c. Tipos de estructura
    i. Tipos definidos por el usuario con el formato struct S {...}
d. Tipos de valor que aceptan valores NULL
    i. Extensiones de todos los demás tipos de valor con un valor null

2. Tipos de referencia
    a. Tipos de clase
        i. Clase base definitiva de todos los demás tipos: object
    b. Cadenas Unicode: string
        i. Tipos definidos por el usuario con el formato class C {...}
    c. Tipos de interfaz
        i. Tipos definidos por el usuario con el formato interface I {...}
    d. Tipos de matriz
        i. Unidimensional y multidimensional; por ejemplo, int[] y int[,]
    e. Tipos delegados
        i. Tipos definidos por el usuario con el formato delegate int D(...)

Los programas de C# utilizan declaraciones de tipos para crear nuevos tipos. Una declaración de tipos especifica el nombre y los miembros del nuevo tipo. Cinco de las categorías de tipos de C# las define el usuario: tipos de clase, tipos de estructura, tipos de interfaz, tipos de enumeración y tipos delegados.

Un tipo class define una estructura de datos que contiene miembros de datos (campos) y miembros de función (métodos, propiedades y otros). Los tipos de clase admiten herencia única y polimorfismo, mecanismos por los que las clases derivadas pueden extender y especializar clases base.

Un tipo struct es similar a un tipo de clase, por el hecho de que representa una estructura con miembros de datos y miembros de función. Sin embargo, a diferencia de las clases, las estructuras son tipos de valor y no suelen requerir la asignación del montón. Los tipos struct no admiten la herencia especificada por el usuario y todos los tipos de struct se heredan implícitamente del tipo object.

Un tipo interface define un contrato como un conjunto con nombre de miembros de función públicos. Un class o struct que implementa una interface debe proporcionar implementaciones de miembros de función de la interfaz. Una interface puede heredar de varias interfaces base, y un class o struct pueden implementar varias interfaces.

Un tipo delegate representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son análogos a los tipos de

función proporcionados por los lenguajes funcionales. Son similares al concepto de punteros de función en otros lenguajes, pero a diferencia de los punteros de función, los delegados están orientados a objetos y presentan seguridad de tipos.

Un tipo enum es un tipo distinto con constantes con nombre. Cada tipo enum tiene un tipo subyacente, que debe ser uno de los ocho tipos enteros. El conjunto de valores de un tipo enum es igual que el conjunto de valores del tipo subyacente.

C# admite matrices unidimensionales y multidimensionales de cualquier tipo. A diferencia de los tipos enumerados anteriormente, los tipos de matriz no tienen que ser declarados antes de usarlos. En su lugar, los tipos de matriz se crean mediante un nombre de tipo entre corchetes. Por ejemplo, int[] es una matriz unidimensional de int, int[,] es una matriz bidimensional de int y int[][] es una matriz unidimensional de la matriz unidimensional de int.

Los tipos de valor NULL tampoco tienen que ser declarados antes de usarlos. Para cada tipo de valor distinto de NULL T, ¿existe un tipo de valor NULL correspondiente T?, que puede tener un valor adicional, null. Por ejemplo, int es un tipo que puede contener cualquier número entero de 32 bits o el valor null.

El sistema de tipos de C# está unificado, de tal forma que un valor de cualquier tipo puede tratarse como un object. Todos los tipos de C# directa o indirectamente se derivan del tipo de clase object, y object es la clase base definitiva de todos los tipos. Los valores de tipos de referencia se tratan como objetos mediante la visualización de los valores como tipo object. Los valores de tipos de valor se tratan como objetos mediante la realización de operaciones de conversión boxing y operaciones de conversión unboxing.

Una **variable** es un elemento del lenguaje que nos permite almacenar un valor de un tipo determinado. En particular, una variable se declara especificando un tipo de datos seguido de un identificador válido y referencia a la zona de memoria donde se almacena cierta cantidad de información del tipo de datos seleccionado. Por ejemplo:

```
int cantidad;
```

Se refiere a una variable denominada cantidad que puede almacenar un valor numérico de tipo entero.

Es posible inicializar el valor de dicha variable en la declaración de la misma de la siguiente forma:

```
int cantidad = 100;
```

Una **constante** es una variable cuyo valor no puedo cambiar durante la ejecución del programa, por lo que el compilador informará con un error de todo intento de modificar el valor inicial de una constante.

Las constantes se definen como variables normales pero precediendo el nombre de su tipo del modificador const y dándoles siempre un valor inicial al declararlas:

```
const int IVA = 21;
```

5. Sentencias o instrucciones

Una **sentencia o instrucción** es una expresión seguida por un punto y coma (;), que utiliza el compilador para separar las diferentes sentencias de un programa. Una sentencia en C# puede contener no sólo expresiones, sino también instrucciones de control de flujo o llamadas a funciones o métodos.

Un programa será entonces un conjunto de sentencias que persiguen un efecto sobre los objetos que modifica. A este conjunto de sentencias se le añaden otros elementos como directivas al compilador, comentarios, etc., que forman el programa en su conjunto.

Una sentencia es nula o vacía cuando sólo contiene el punto y coma (;).

Cuando varias sentencias se agrupan entre llaves ({ }) forman un bloque, sintácticamente equivalente a una sentencia. Los bloques de sentencias se pueden anidar y pueden llevar sus propias declaraciones.

5.1. Sentencia de asignación

Antes de utilizar una variable, debemos declararla. Al declararla podemos especificar su nombre y características. El nombre de la variable es referido como un identificador.

El lenguaje C# tiene reglas específicas relacionadas con el uso de los identificadores:

- Un identificador solo puede contener letras, dígitos y el carácter guion bajo.
- Un identificador debe iniciar con una letra o un guion bajo.
- Un identificador no debería ser una de las palabras clave que visual C# reserva para su propio uso.
- C# es sensible a mayúsculas y minúsculas.

Al declarar una variable, debemos elegir un nombre que tenga significado respecto a lo que almacena, de esta forma, el código será más fácil de entender.

Cuando se declara una variable, se reserva un espacio de almacenamiento en memoria para esa variable y el tipo de datos que va a contener. Podemos declarar múltiples variables en una sola declaración utilizando el separador coma, todas las variables declaradas de esta manera, son del mismo tipo de datos.

```
int cantidad;
```

Después de declarar la variable, podemos asignarle un valor utilizando una operación de asignación. Durante la ejecución de la aplicación, podemos cambiar el valor de una variable tantas veces como queramos. El operador de asignación = nos permite asignar un valor a una variable. También es posible declarar una variable y asignar su valor al mismo tiempo.

```
int cantidad = 100;
```

Los tipos básicos de datos admiten los **modificadores de tipo** que permiten una modificación del rango de valores, aplicando las siguientes palabras reservadas:

- **short:** corto.
- **long**: largo.
- **signed**: con signo.
- **unsigned**: sin signo.

Los modificadores se aplican a los tipos int y char, a excepción del modificador long que también se puede aplicar al tipo double.

Existen también cuatro **modificadores de acceso** (public, protected, internal y private) que se pueden emplearse para controlar el modo en el que las variables se modifican en un programa. Así, pueden especificarse los siguientes seis niveles de accesibilidad con los modificadores de acceso:

- **public**: El acceso no está restringido.
- **protected**: El acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora.
- **internal**: El acceso está limitado al ensamblado actual.
- **protected internal**: El acceso está limitado al ensamblado actual o a los tipos derivados de la clase contenedora.
- **private**: El acceso está limitado al tipo contenedor.
- **private protected**: El acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora que hay en el ensamblado actual.

5.2. Sentencias de selección

El lenguaje C# proporciona ciertas sentencias de selección o decisión que permiten controlar si una instrucción o conjunto de instrucciones se deben ejecutar o no según la evaluación de una condición determinada.

La **sentencia if-else** tiene el siguiente formato general:

```
if (<expresión>) {
   sentencias_si_verdadero;
}
[ else {
   sentencias_si_falso;
} ]
```

La cláusula **else** es opcional. La expresión debe dar un valor verdadero o falso. Durante la ejecución de programa se evalúa la expresión, que en caso de ser verdadera da lugar a que se ejecute la sentencia_verdadera. Si da un valor falso se ejecuta la instrucción siguiente a la estructura if, a menos que aparezca la palabra reservada else, en cuyo caso se ejecutaría la sentencia_falsa.

Si sentencia_verdadera o sentencia_falsa no fuera una sola sentencia y estuviera compuesta por un grupo de ellas, se deberán delimitar por una llave abierta ({) y una llave cerrada (}).

La estructura if se puede anidar con otras estructuras if.

En el ejemplo siguiente, la variable booleana condition se establece en true y, a continuación, se comprueba en la instrucción if . El resultado es “La variable se ha inicializado a Verdadero”:

```
bool condition = true;

if (condition) {
    Console.WriteLine("La variable se ha inicializado a Verdadero.");
}
else {
    Console.WriteLine("La variable se ha inicializado a falso.");
}
```

La **sentencia switch** toma una decisión para cada una de las posibilidades contempladas. El formato general de esta estructura es el siguiente:

```
switch (<variable>) {
```

```
    case <valor1>:
        sentencias1;
        break;
    case <valor2>:
        sentencias2;
        break;
    ...
    default:
        sentencias;
        break;
}
```

Hay que tener en cuenta que la comparación del valor de la variable, que debe ser de tipo int o char, con cada valor se realiza sólo por igualdad y no se evalúan expresiones.

El siguiente fragmento de código evalúa el valor de la variable caseSwitch y muestra por pantalla el texto "Caso 1":

```
int caseSwitch = 1;

switch (caseSwitch) {
    case 1:
        Console.WriteLine("Caso 1");
        break;
    case 2:
        Console.WriteLine("Caso 2");
        break;
    default:
        Console.WriteLine("Caso por defecto ");
        break;
}
```

5.3. Sentencias de iteración

El lenguaje C# proporciona ciertas sentencias de iteración o repetición que permiten ejecutar una instrucción o conjunto de instrucciones un número determinado de veces según la evaluación de una condición.

La **sentencia for** que tiene el siguiente formato general:

```
for (exp_inicialización ; exp_condición ; exp_incremento) {
sentencias;
}
```

La exp_inicialización es una expresión simple, normalmente una asignación a una variable de control del bucle de un valor inicial. La exp_condición es una expresión lógica o de comparación que determina el final del bucle. La exp_incremento es una expresión simple que modifica la variable de control al final de cada vuelta o iteración. La variable de control también se puede decrementar.

Lo primero que se evalúa es la exp_inicialización, a continuación la exp_condición del bucle, que en caso de ser falsa da lugar a que termine la ejecución del mismo. Si es verdadera se ejecuta la sentencia o bloque de sentencias. Después de ejecutarse el cuerpo del bucle se evalúa la exp_incremento, alterando la variable de control.

En el siguiente ejemplo se muestra la instrucción for con todas las secciones definidas:

```
for (int i = 0; i < 5; i++) {
    Console.WriteLine(i);
}
```

De la misma forma se pueden tener bucles infinitos, que son aquellos que, en principio, no tienen fin:

```
for (;;) {
    Console.WriteLine("Hola ");
}
```

La **sentencia while** tiene el siguiente formato general:

```
while (<exp_condición>) {
   sentencias;
}
```

Se compone de la palabra reservada while seguida de la expresión que controla el bucle y el cuerpo de la estructura, que consiste en una instrucción o bloque de instrucciones que se ejecutan en cada iteración. La exp_condición es cualquier expresión simple que al evaluarse devuelve verdadero o falso. El bucle se repite mientras la condición sea verdadera. Cuando es falsa el programa pasa a la siguiente instrucción que sigue después del cuerpo de la estructura while. La condición de salida se evalúa antes de cada vuelta o iteración.

En el ejemplo siguiente se muestra el uso de la instrucción while:

```
int n = 0;

while (n < 5) {
    Console.WriteLine(n);
    n++;
}
```

La **sentencia do-while** repite la ejecución de una sentencia o bloque de sentencias hasta que se cumpla una condición determinada. Para ello evalúa la expresión de control del bucle después de que el cuerpo de la estructura se ejecute. El formato general de esta estructura es el siguiente:

```
do {
   sentencias;
} while (<exp_condición>);
```

Si el resultado de exp_condición es verdadero se ejecuta de nuevo el cuerpo del bucle y así sucesivamente hasta que la condición sea falsa. Cuando el cuerpo del bucle es una sola sentencia las llaves no son obligatorias.

En el ejemplo siguiente se muestra el uso de la instrucción do:

```
int n = 0;
```

```
do {
    Console.WriteLine(n);
    n++;
} while (n < 5);
```

La **sentencia foreach** ejecuta una instrucción o un bloque de instrucciones para cada elemento en una instancia del tipo que implementa la interfaz System.Collections.IEnumerable o System.Collections.Generic.IEnumerable<T>. La instrucción foreach no se limita a esos tipos y puede aplicarse a una instancia de cualquier tipo que satisfaga las siguientes condiciones:

- tiene el método público sin parámetros GetEnumerator cuyo tipo de valor devuelto es clase, estructura o tipo de interfaz,
- el tipo de valor devuelto del método GetEnumerator tiene la propiedad pública Current y el método público sin parámetros MoveNext cuyo tipo de valor devuelto es Boolean.

El siguiente ejemplo muestra el uso de la instrucción foreach con una instancia del tipo List<T> que implementa la interfaz IEnumerable<T>:

```
var fibNumbers = new List<int> {0, 1, 1, 2, 3, 5, 8, 13};
int count = 0;

foreach (int element in fibNumbers) {
    count++;
    Console.WriteLine($"Elemento nº{count}: {element}");
}

Console.WriteLine($"Número de elementos: {count}");
```

La **sentencia break** se utiliza para salir de un bucle antes de la finalización normal del mismo o por alguna condición especial. Cuando se ejecuta la sentencia break, el control del programa pasa inmediatamente a la instrucción siguiente al cuerpo del bucle. También se emplea para salir de un tratamiento de la estructura switch.

La **sentencia continue** se puede utilizar dentro de las estructuras for, while y do-while donde fuerza, al ejecutarse, a que comience una nueva iteración o vuelta dentro de la estructura repetitiva correspondiente. Dentro del while o del do-while se evalúa la condición de salida. En el for se ejecuta un salto a la expresión de incremento o decremento.

6. Enumeraciones

Un tipo de enumeración es un tipo de valor distinto con un conjunto de constantes con nombre. Las enumeraciones se definen cuando se necesita fijar un tipo que pueda tener un conjunto de valores discretos. Usan uno de los tipos de valor integral como almacenamiento subyacente. Proporcionan significado semántico a los valores discretos.

En el ejemplo siguiente se declara y usa un tipo enum denominado Color con tres valores constantes, Red, Green y Blue:

```
using System;

enum Color {
    Red,
    Green,
    Blue
}
```

```
class EnumExample {

    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }

}
```

A cada tipo enum le corresponde un tipo entero conocido como el tipo subyacente del tipo enum. Un tipo enum que no declara explícitamente un tipo subyacente tiene un tipo subyacente de int. El formato de almacenamiento de un tipo enum y el intervalo de valores posibles se determinan por el tipo subyacente.

En el ejemplo siguiente se declara un tipo enum denominado Alignment con un tipo subyacente de sbyte:

```
enum Alignment : sbyte {
    Left = -1,
    Center = 0,
    Right = 1
}
```

Como se muestra en el ejemplo anterior, una declaración de miembro enum puede incluir una expresión constante que especifica el valor del miembro. El valor constante para cada miembro enum debe estar en el intervalo del tipo subyacente de enum. Cuando una declaración de miembro enum no especifica explícitamente un valor, al miembro se le asigna el valor cero, si es el primer miembro en el tipo enum, o el valor del miembro textualmente anterior enum más uno.

7. Estructuras

Al igual que las clases, las **estructuras** o **structs** son conjuntos de datos que pueden contener miembros de datos y miembros de función, pero a diferencia de las clases, los structs son tipos de valor y no requieren asignación del heap (montón).

Una variable de un tipo de struct almacena directamente los datos del struct, mientras que una variable de un tipo de clase almacena una referencia a un objeto asignado dinámicamente en el heap.

Los tipos struct no admiten la herencia y se heredan implícitamente del tipo ValueType, que a su vez se hereda implícitamente de object.

Los structs son particularmente útiles para estructuras de datos pequeñas que tengan semánticas de valor. Los números complejos, los puntos de un sistema de coordenadas o los pares clave-valor de un diccionario son buenos ejemplos de structs. El uso de un struct en lugar de una clase para estructuras de datos pequeñas puede suponer una diferencia sustancial en el número de asignaciones de memoria que realiza una aplicación.

Por ejemplo, el siguiente programa crea e inicializa una matriz de 100 puntos. Si Point se implementa como una clase, se crean instancias de 101 objetos distintos: uno para la matriz y uno por cada uno de los 100 elementos:

```
public class PointExample {
    public static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++)
            points[i] = new Point(i, i);
    }
}
```

Una alternativa es convertir Point en un struct:

```
struct Point {
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Ahora, se crea la instancia de un solo objeto: la de la matriz, y las instancias de Point se asignan en línea dentro de la matriz.

Los constructores structs se invocan con el operador new, similar a un constructor de clase. Sin embargo, en lugar de asignar dinámicamente un objeto en el heap o montón administrado y devolver una referencia a él, un constructor de structs simplemente devuelve el valor del struct propiamente dicho, normalmente en una ubicación temporal en la pila, y este valor se copia luego cuando es necesario.

8. Matrices

Una **matriz** o **array** es una estructura de datos que contiene un número de variables a las que se accede mediante índices calculados. Las variables contenidas en una matriz, denominadas también elementos de la matriz, son todas del mismo tipo y este tipo se conoce como tipo de elemento de la matriz.

Los tipos de matriz son tipos de referencia, y la declaración de una variable de matriz simplemente establece un espacio reservado para una referencia a una instancia de matriz. Las instancias de matriz reales se crean dinámicamente en tiempo de ejecución mediante el operador new. La nueva operación especifica la longitud de la nueva instancia de matriz, que luego se fija para la vigencia de la instancia. Los índices de los elementos de una matriz van de 0 a Length - 1. El operador new inicializa automáticamente los elementos de una matriz a su valor predeterminado, que, por ejemplo, es cero para todos los tipos numéricos y null para todos los tipos de referencias.

En el ejemplo siguiente se crea una matriz de elementos int, se inicializa la matriz y se imprime el contenido de la matriz:

```
using System;

class ArrayExample {

    static void Main() {
        int[] a = new int[10];

        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }

        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine($"a[{i}] = {a[i]}");
        }
    }

}
```

Este ejemplo se crea y se pone en funcionamiento en una matriz unidimensional. C# también admite matrices multidimensionales. El número de dimensiones de un tipo de matriz, conocido también como **rango** del tipo de matriz, es una más el número de comas escritas entre los corchetes del tipo de matriz. En el ejemplo siguiente se asignan una **matriz unidimensional**, **multidimensional** y **tridimensional**, respectivamente:

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

La matriz a1 contiene 10 elementos, la matriz a2 50 elementos (10 × 5) elementos y la matriz a3 100 elementos (10 × 5 × 2) elementos. El tipo de elemento de una matriz puede ser cualquiera, incluido un tipo de matriz.

Una matriz con elementos de un tipo de matriz a veces se conoce como **matriz escalonada** porque las longitudes de las matrices de elementos no tienen que ser iguales. En el ejemplo siguiente se asigna una matriz de matrices de int:

```
int[][] a = new int[3][];

a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

La primera línea crea una matriz con tres elementos, cada uno de tipo int[] y cada uno con un valor inicial de null. Las líneas posteriores inicializan entonces los tres elementos con referencias a instancias de matriz individuales de longitud variable.

El nuevo operador permite especificar los valores iniciales de los elementos de matriz mediante un inicializador de matriz, que es una lista de las expresiones escritas entre los delimitadores { y }. En el ejemplo siguiente se asigna e inicializa un tipo int[] con tres elementos:

```
int[] a = new int[] { 1, 2, 3 };
```

Las declaraciones de variable local y campo se pueden acortar más para que así no sea necesario reformular el tipo de matriz:

```
int[] a = { 1, 2, 3 };
```

Los dos ejemplos anteriores son equivalentes a lo siguiente:

```
int[] t = new int[3];

t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

9. Colecciones

Para muchas aplicaciones, puede que desee crear y administrar grupos de objetos relacionados. Existen dos formas de agrupar objetos: mediante la creación de matrices de objetos y con la creación de colecciones de objetos.

Las matrices son muy útiles para crear y trabajar con un número fijo de objetos fuertemente tipados.

Las **colecciones** proporcionan una manera más flexible de trabajar con grupos de objetos. A diferencia de las matrices, el grupo de objetos con el que trabaja puede aumentar y reducirse de manera dinámica a medida que cambian las necesidades de la aplicación. Para algunas colecciones, se puede asignar una clave a cualquier objeto que incluya en la colección para, de este modo, recuperar rápidamente el objeto con la clave. Una colección es una clase, por lo que debe declarar una instancia de la clase para poder agregar elementos a dicha colección.

Si la colección contiene elementos de un solo tipo de datos, se puede usar una de las clases del espacio de nombres System.Collections.Generic. Una colección genérica cumple la seguridad de tipos para que ningún otro tipo de datos se pueda agregar a ella. Cuando se recupera un elemento de una colección genérica, no se tiene que determinar su tipo de datos ni convertirlo.

En el ejemplo siguiente se crea una lista de cadenas utilizando la clase genérica List<T>, que permite trabajar con una lista de objetos fuertemente tipados, y luego se recorren en iteración mediante una instrucción foreach:

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Si el contenido de una colección se conoce de antemano, puede usar un inicializador de colección para inicializar la colección.

El ejemplo siguiente es el mismo que el ejemplo anterior, excepto que se usa un inicializador de colección para agregar elementos a la colección:

```
// Create a list of strings by using a collection initializer.
var salmons = new List<string> {"chinook", "coho", "pink", "sockeye"};

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

El ejemplo siguiente quita un elemento de la colección especificando el objeto que se quitará:

```
// Create a list of strings by using a collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

.NET Framework proporciona muchas colecciones comunes. Cada tipo de colección está diseñado para un fin específico. A continuación, se describen algunas de las clases de colecciones más comunes:

- ✓ Clase System.Collections.Generic.
- ✓ Clase System.Collections.Concurrent.
- ✓ Clase System.Collections.

Una colección genérica **System.Collections.Generic** es útil cuando todos los elementos de la colección tienen el mismo tipo. Una colección genérica exige el establecimiento de fuertes tipos al permitir agregar solo los tipos de datos deseados.

A continuación, se enumeran algunas de las clases de System.Collections.Generic:

Clase	Descripción
Dictionary<TKey,TValue>	Representa una colección de pares de clave y valor que se organizan según la clave.
List<T>	Representa una lista de objetos a los que puede tener acceso el índice. Proporciona métodos para buscar, ordenar y modificar listas.
Queue<T>	Representa una colección de objetos de primeras entradas, primeras salidas (FIFO).
SortedList<TKey,TValue>	Representa una colección de pares clave-valor que se ordenan por claves según la implementación de IComparer<T> asociada.
Stack<T>	Representa una colección de objetos de últimas entradas, primeras salidas (LIFO).

En .NET Framework 4 o posterior, las colecciones del espacio de nombres **System.Collections.Concurrent** proporcionan operaciones eficaces y seguras para subprocesos con el fin de obtener acceso a los elementos de la colección desde varios subprocesos.

Las clases del espacio de nombres System.Collections.Concurrent deben usarse en lugar de los tipos correspondientes de los espacios de nombres System.Collections.Generic y System.Collections cada vez que varios subprocesos tengan acceso de manera simultánea a la colección.

Algunas clases incluidas en el espacio de nombres System.Collections.Concurrent son BlockingCollection<T>, ConcurrentDictionary<TKey,TValue>, ConcurrentQueue<T> y ConcurrentStack<T>.

Las clases del espacio de nombres **System.Collections** no almacenan los elementos como objetos de un tipo específico, sino como objetos del tipo Object.

Siempre que sea posible, se deben usar las colecciones genéricas del espacio de nombres System.Collections.Generic o del espacio de nombres System.Collections.Concurrent en lugar de los tipos heredados del espacio de nombres System.Collections.

En la siguiente tabla se enumeran algunas de las clases usadas con frecuencia en el espacio de nombres System.Collections:

Clase	Descripción
ArrayList	Representa una matriz cuyo tamaño aumenta dinámicamente cuando es necesario.
Hashtable	Representa una colección de pares de clave y valor que se organizan por código hash de la clave.
Queue	Representa una colección de objetos de primeras entradas, primeras salidas (FIFO).
Stack	Representa una colección de objetos de últimas entradas, primeras salidas (LIFO).