Universidad Nacional Autónoma de México
Programa de Tenología en Cómputo
Lo que siempre quisiste saber de C# y nunca te atreviste a preguntar

Elaborado por:

Rivera Negrete Manuel Armando

28 de Junio del 2018

Índice general

¿Por qué aprender C#?

Parte I

C# Básico

Capítulo 1

La arquitectura .NET

1.1. C# y su lugar dentro de .NET

C# (leído «C sharp») es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como partes de su plataforma .NET. Aunque esta plataforma permite desarrollar aplicaciones en otros lenguajes de programación, C# ha sido creado específicamente para .NET, adecuando todas sus estructuras a las características y capacidades de dicha plataforma.
.NET es un framework de Microsoft que hace un énfasis en el desarrollo sencillo de aplicaciones, independencia de hardware y transparencia de redes. Es una implementación de Common Language Infraestructure (Estandar CLI). Un Framework es un esquema (un esqueleto, un patrón) para el desarrollo y/o la implementación de una aplicación.

1.2. Common Language Runtime

El Common Language Runtime o CLR es un entorno de ejecución que ejecuta el código y proporciona servicios que facilitan el proceso de desarrollo de los programas que corren sobre la plataforma Microsoft .NET.
Los compiladores y las herramientas exponen la funcionalidad del tiempo de ejecución del idioma común y le permiten escribir código que se beneficia de este entorno de ejecución administrada. El código que desarrolla con un compilador de lenguaje que se dirige al tiempo de ejecución se denomina código administrado; se beneficia de características tales como la integración entre idiomas, manejo de excepciones entre idiomas, seguridad mejorada, soporte de versiones e implementación, un modelo simplificado para la interacción de componentes y servicios de depuración y creación de perfiles. Para enternder

el proceso de compilación en C# es necesario definir algunos conceptos:
Compilación: La tarea de compilar se refiere al proceso de traducción del código fuente a código entendible por la computadora, entendiéndose por código fuente las líneas de código que se han escrito en un lenguaje de programación, en este caso un lenguaje de programación de alto nivel.
Código Máquina: Es el sistema de códigos directamente interpretable por un circuito microprogramable, como el microprocesador de una computadora, es decir, es un lenguage que entiende la computadora.
ByteCode: Es un código intermedio más abstracto que el código máquina. Habitualmente es tratado como un archivo binario que contiene un programa ejecutable similar a un módulo objeto, que es un archivo binario producido por el compilador cuyo contenido es el código objeto o código máquina.

1.3. Microsoft Intermediate Language

MSIL significa Microsoft Intermediate Language. Podemos llamarlo Lenguaje Intermedio (IL) o Lenguaje Intermedio Común (CIL). Durante el tiempo de compilación, el compilador convierte el código fuente en Microsoft Intermediate Language (MSIL). Microsoft Intermediate Language (MSIL) es un conjunto de instrucciones independiente de la CPU que se puede convertir de manera eficiente al código nativo. Durante el tiempo de ejecución, el compilador Just In Time (JIT) de Common Language Runtime (CLR) convierte el código de Microsoft Intermediate Language (MSIL) en código nativo al sistema operativo.
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. Luego, 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 çódigo administrado", a diferencia del çódigo no administra-

do"que se compila en lenguaje de máquina nativo destinado a un sistema específico. En el siguiente diagrama 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.
Ninguno de los compiladores que generan código para la plataforma .NET produce código máquina para CPUs x86 ni para ningún otro tipo de CPU concreta, sino que generan código escrito en el lenguaje intermedio conocido como Microsoft Intermediate Lenguage (MSIL) El CLR da a las aplicaciones la sensación de que se están ejecutando sobre una máquina virtual, y precisamente MSIL es el código máquina de esa máquina virtual. Es decir, MSIL es el único código que es capaz de interpretar el CLR, y por tanto cuando se dice que un compilador genera código para la plataforma .NET lo que se está diciendo es que genera MSIL.
MSIL ha sido creado por Microsoft tras consultar a numerosos especialistas en la escritura de compiladores y lenguajes tanto del mundo académico como empresarial. Es un lenguaje de un nivel de abstracción mucho más alto que el de la mayoría de los códigos máquina de las CPUs existentes, e incluye instrucciones que permiten trabajar directamente con objetos (crearlos, destruirlos, inicializarlos, llamar a métodos virtuales, etc.), tablas y excepciones (lanzarlas, capturarlas y tratarlas).

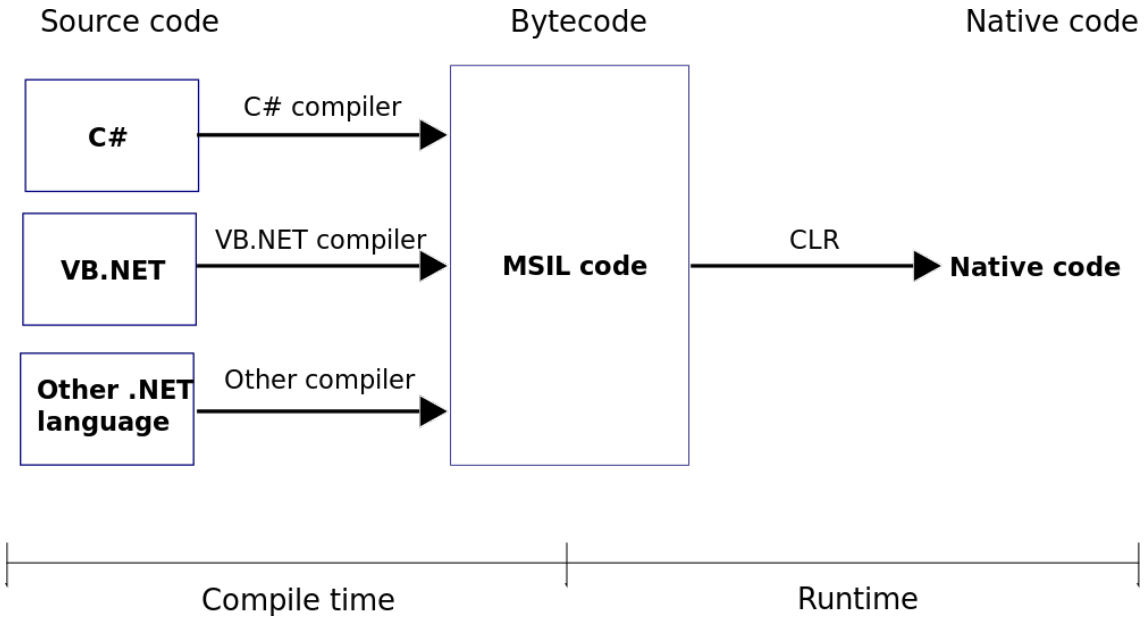

Figura 1.1: Compilación en C#

1.4. Assemblies

Los assemblies son los bloques de construcción de las aplicaciones del framework .NET. Un assembly es una colección de tipos y recursos que están construidos para trabajar en conjunto y formar la unidad lógica de la funcionalidad del programa.
Todos los tipos en .NET Framework deben existir en assemblies; el tiempo de ejecución de idioma común no admite tipos fuera de ensamblados. Cada vez que crea una aplicación de Microsoft Windows, un servicio de Windows, una biblioteca de clases u otra aplicación con Visual Basic .NET, está creando un solo assembly. Cada assembly se almacena como un archivo .exe o .dll.
Aunque es técnicamente posible crear assemblies que abarquen múltiples archivos, no es probable que use esta tecnología en la mayoría de las situaciones.
.NET Framework usa assemblies como la unidad fundamental para varios propósitos:
•Seguridad
•Tipo de identidad
•Versiones
•Desarrollo

Capítulo 2

Introducción a Visual Studio 2017 y C#

2.1. Características de C#

C# es un lenguaje de programación orientado a objetos, al ser posterior a C++ y Java. los lenguajes de programación orientados a objetos más conocidos hasta entonces, C# combina y mejora gran parte de las características más interesantes de ambos lenguajes. Por tanto, un programador que conozca C# a fondo no tendrá problemas para programar tanto en C++ como en Java, sus anteccesores.
El nombre fue inspirado por la notación musical '#' (llamada sostenido, en inglés Sharp) que indica que la nota es de un tono más alto. Se puede utilizar C# para crear aplicaciones cliente de Windows, servicios Web XML, componentes distribuidos, aplicaciones cliente-servidor, aplicaciones de base de datos, y mucho, mucho más.
Para poder crear programas en C# y ejecutarlos posteriormente, es necesario tener instalado en el PC los siguientes paquetes:

.NET Framework SDK: Es el kit de desarrollo e incluye un compilador de línea de C# y bibliotecas que contienen una amplia colección de clases previamente definidas que podemos utilizar en nuestras aplicaciones; es decir, contiene todo lo necesario para poder crear y compilar nuestros programas.
.NET Framework Redistributable Package: Permite la ejecución de programas creados en C#. Esto es necesario porque la compilación de C# no genera, como habitualmente para otros lenguajes, código máquina, sino un código escrito en un lenguaje propio de Microsoft: MSIL (Microsoft Intermediate Language). El CLR (Common Language Runtime) es el núcleo de

la plataforma .NET y se encarga de gestionar la ejecución de los programas escritos en MSIL, ambos se pueden descargar gratuitamente desde la página web de la Red de Desarrolladores para Microsoft (Microsoft Developers Network) :
https://www.microsoft.com/es-mx/download

También pueden crearse programas mediante la herramienta Visual Studio(Incluye en su instalación dichos paquetes), que ofrece un interfaz gráfico muy amigable y cómodo de utilizar, esta herramienta cuenta con una versión de paga y una gratuita (Community) y se puede descargar directamente desde la página oficial.
https://visualstudio.microsoft.com/es/

En la figura 2.2 se muestra un menú de paquetes complementarios que podemos descargar, si sólo queremos desarrollar en C# basta con seleccionar los primeros dos paquetes.

Figura 2.1: Página oficial para descargar Visual Studio

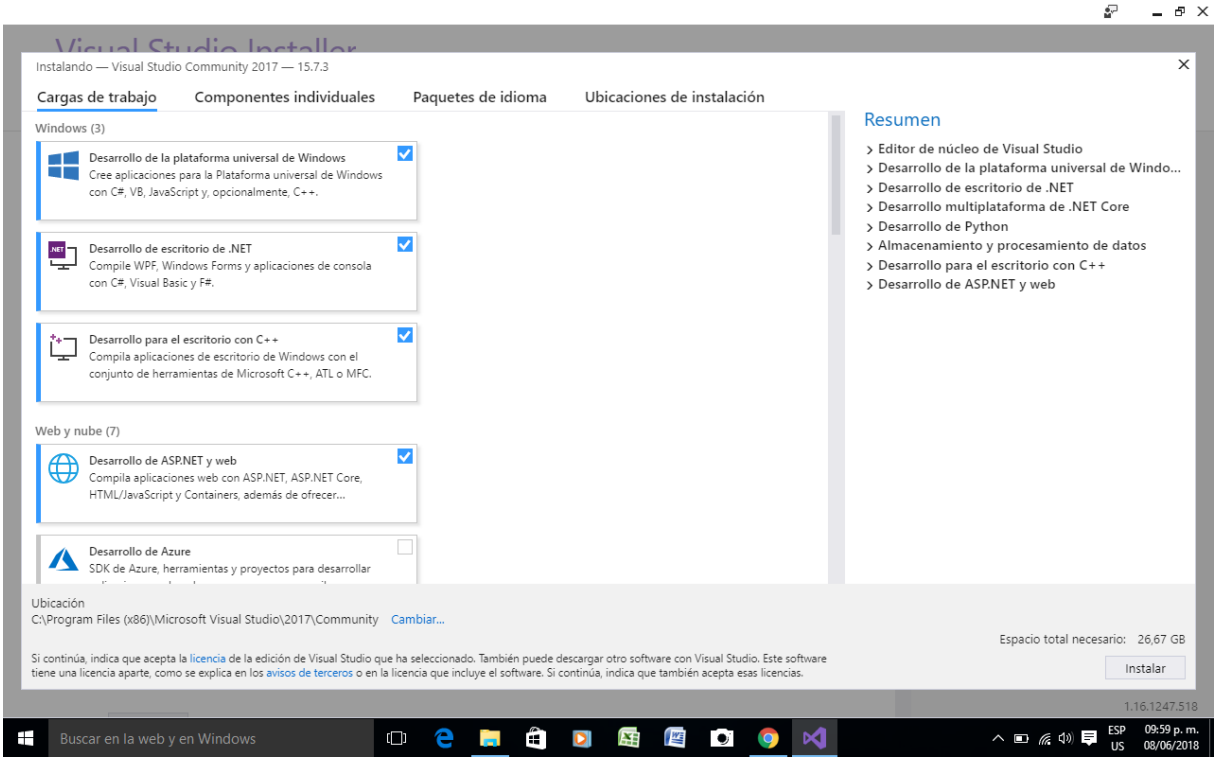
Visual Studio Installer
Instalando — Visual Studio Community 2017 — 15.7.3
Cargas de trabajo
Componentes individuales
Paquetes de idioma
Ubicaciones de instalación
Windows (3)
Desarrollo de la plataforma universal de Windows
Cree aplicaciones para la Plataforma universal de Windows con C#, VB, JavaScript y, opcionalmente, C++.
Desarrollo de escritorio de .NET
Compile WPF, Windows Forms y aplicaciones de consola con C#, Visual Basic y F#.
Desarrollo para el escritorio con C++
Compila aplicaciones de escritorio de Windows con el conjunto de herramientas de Microsoft C++, ATL o MFC.
Web y nube (7)
Desarrollo de ASP.NET y web
Compila aplicaciones web con ASP.NET, ASP.NET Core, HTML/JavaScript y Containers, además de ofrecer...
Desarrollo de Azure
SDK de Azure, herramientas y proyectos para desarrollar
Resumen
Editor de núcleo de Visual Studio
Desarrollo de la plataforma universal de Windo...
Desarrollo de escritorio de .NET
Desarrollo multiplataforma de .NET Core
Desarrollo de Python
Almacenamiento y procesamiento de datos
Desarrollo para el escritorio con C++
Desarrollo de ASP.NET y web
Ubicación
C:\Program Files (x86)\Microsoft Visual Studio\2017\Community
Cambiar...
Si continúa, indica que acepta la licencia de la edición de Visual Studio que ha seleccionado. También puede descargar otro software con Visual Studio. Este software tiene una licencia aparte, como se explica en los avisos de terceros o en la licencia que incluye el software. Si continúa, indica que también acepta esas licencias.
Espacio total necesario: 26,67 GB
Instalar
1.16.1247.518
Buscar en la web y en Windows
ESP
US
09:59 p. m.
08/06/2018

Figura 2.2:

2.2. Estructura de un programa

Al abrir Visual Studio en la parte superior podemos observar un menú, para empezar un nuevo proyecto hacemos clic en la pestaña de Archivo, posteriormente seleccionamos la opción de nuevo y después proyecto, también podemos presionar la combinación de teclas Ctrl+Mayus+N. Se nos abrirá una ventana en la cual seleccionaremos la opción « Aplicacion de Consola » y procedemos a darle un nombre a nuestro proyecto y damos click en aceptar. Ver las figuras 2.3 y 2.4.

Para crear un proyecto sin Visual Studio necesitamos un editor de texto

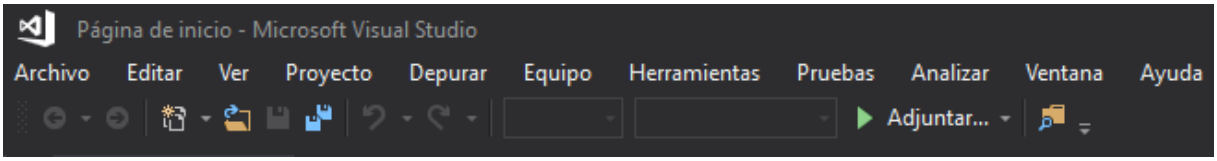

Figura 2.3: Menú superior en Visual Studio

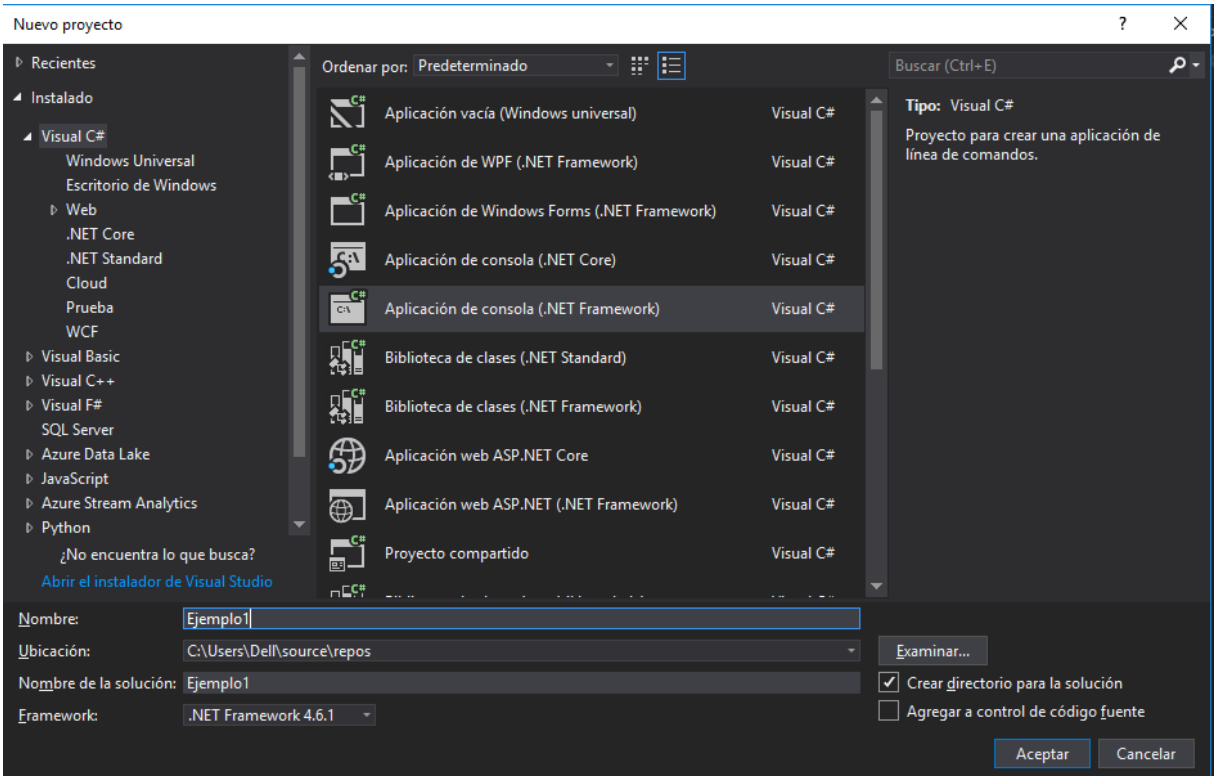

Figura 2.4:

plano y los paquetes antes mencionados. Creamos una carpeta en el escrito llamada csharp, abrimos el block de notas y en este escribimos el código del ejemplo 1 con la extension *.cs* y los guardamos dentro de la carpeta. Para poder compilar y ejecutar este programa necesitamos tener el comando *csc* (el compilador de C# incluido en la plataforma .NET) accesible. Para ello hay que modificar la variable de entorno Path, que contiene las carpetas en las que el sistema busca los programas a los que se invoca desde la línea de comandos. El proceso es el siguiente:

1.- Buscar el lugar donde está instalada nuestra versión de .NET El Lugar por defecto es en una carpeta de la forma vXXXXX (donde XXXXX representa el número de versión de .NET que hemos instalado) que puede encontrarse en:
C:\Windows\Microsoft.NET\Framework\vX.X.XXXXX

2.- Abrir la cmd, esto se puede hacer abriendo el explorador y poner cmd o presionar la combinacion de las teclas Windows + R y escribir cmd. En la ventana de comandos escribir:
path=%path%;C:\Windows\Microsoft.NET\Framework\vX.X.XXXXX.
El comando *csc* ya debe estar accesible desde la línea de comandos.

C# es un lenguaje orientado a objetos. En cualquier programa en C# debe existir al menos una clase que contenga un método llamado **Main**. Este método constituye lo que se denomina punto de entrada, y define por dónde ha de comenzar a ejecutarse la aplicación: la primera instrucción ejecutada será la primera instrucción del método **Main**.
El siguiente ejemplo muestra el programa más simple que puede crearse en C#.

Ejemplo 2.1

```
1 using System;
2 //Usamos el espacio de nombres System
3 class Ejemplo1 {
4   static void Main() {
5
6   }
7 }
```

Para crear una clase hay que escribir la palabra reservada **class** seguida del nombre que queremos darle. A continuación entre llaves, aparecerán los

métodos y atributos de dicha clase. En este ejemplo hemos creado la clase *Ejemplo 2.1*, que no incluye ningún atributo y contiene un solo método, de nombre **Main**. Como hemos dicho, todo programa en C# debe contener al menos una clase con un método llamado **Main**.

Es importante aclarar que C# se distinguen las mayúsculas de las minúsculas, algo que no curre en todos los lenguajes de programación. Así, el punto de entrada ha de llamarse *Main* y no *main* o *MAIN* o ninguna otra variante.

Como para cualquier otro método existen varias alternativas válidas oara crear la declaración del método **Main**. Algunas son:

public static int Main()
public static void Main(string[] args)
static int Main(string[] args)

La palabra **public** indica que el método es público, es decir, puede ser utilizado por otra clase (si no se pone nada, se considerará privado por defecto). La palabra **static** indica que que el método está asociado a la clase a la que pertenece y no a los objetos que se creen de dicha clase. En tercer lugar, aparece el tipo de la información que devuelve el método: **int** indica que devuelve un dato de tipo entero; **void** indica que no se devuelve ningún valor. A continuación del nombre del método, que para el punto de entrada siempre es **Main**, aparecen entre paréntesis los argumentos o datos de entrada de este método, es decir, la información de la partida que requiere. Esta sección puede estar vacía.

Por el momento dado los escasos conocimientos que aún tenemos, será suficiente con que el punto de entrada sea estático, no tenga argumentos y no devuelva ningún valor, tal y como se declaró en el ejemplo.

2.3. Compilación y ejecución

La compilación y ejecución usando Visual Studio resulta ser muy sencilla sin embrago se explicará el proceso con dicha herramienta y sin ella.
Para compilar nos vamos al menú que se encuentra en la parte superior de la ventana de C# y damos click en la pestaña donde dice Compilar, posteriormente se nos abrirá un menú en el cual seleccionaremos la opción de compilar solución. Al hacer dicho paso en la parte inferior aparecerá otra ventana conocida como la ventana de salida o de output donde se nos infor-

mará de todos los errores en nuestro programa antes de ejecutarlo. Una vez compilado nuestro programa nos vamos a la ventana superior y damos click en la pestaña donde dice depurar, al abrirse el menú seleccionamos la opción de iniciar sin depurar y se nos abrirá la ventana de comandos con nuestro programa en ejecución.
Para compilar un programa sin Visual Studio nos vamos a la carpeta csharp creada en la sección pasada desde la cmd, para hacer esto abrimos la ventana de comandos y escribimos **cd c:\csharp** y pulsamos intro, despues escribimos **csc Ejemplo1.cs** si sólo arroja un mensaje en el que nos indique la versión del compilador que se está utilizando, así como la version del Framework que está instalada significa que el código ha sido compilado con éxito, para ejecutar nuestro programa escribimos el nombre de éste sin la extensión .cs.

Capítulo 3

Tipos predefinidos y control de flujo

En este capitulo se mostrará cómo se utilizan datos de diferentes tipos básicos dentro de un programa y qué tipo de operaciones pueden realizarse con ellos.
Normalmente, los programas sencillos necesitan de datos muy sencillos, pero los que conocemos hasta ahora son insuficientes. Por ejemplo, no podríamos plantearnos un programa que multiplique dos números enteros.

Para manejar datos de tipos básicos cono enteros o reales, necesitamos dos cosas: poder almacenar esos datos en algún sitio, para lo que se utilizan variables, y poder manipular esos datos, para lo que necesitamos los operadores. Veremos a continuación qué tipos de datos tenemos disponibles en C# y cuál es el conjunto de operadores que nos va a permitir transformar y operar con dichos datos.

3.1. Tipos predefinidos de C#

El concepto de dato está relacionado con las operaciones que se pueden realizar sobre él. Un tipo de dato queda definido como un conjunto de valores que tienen asociadas una serie de operaciones para crearlos y manipularlos.

En el ordenador cada tipo de datos se representa de una forma diferente. Una tabla parcial de los tipos que pueden manejarse en C# se muestra en la figura 3.1

El nombre del tipo y la clase donde se define son, en realidad, la misma cosa. Es decir, el nombre del tipo es simplemente un alias y podemos utilizar

Tipo de C#	Tipo de .NET Framework
bool	`System.Boolean`
byte	`System.Byte`
sbyte	`System.SByte`
char	`System.Char`
decimal	`System.Decimal`
double	`System.Double`
float	`System.Single`
int	`System.Int32`
uint	`System.UInt32`
long	`System.Int64`
ulong	`System.UInt64`
object	`System.Object`
short	`System.Int16`

Figura 3.1: Resumen del sistema de Tipos de C#

Operadores lógicos, condicionales y NULL

Categoría	Expresión	Description
AND lógico	x & y	AND bit a bit entero, AND lógico booleano
XOR lógico	x ^ y	XOR bit a bit entero, XOR lógico booleano
OR lógico	x | y	OR bit a bit entero, OR lógico booleano
AND condicional	x && y	Evalúa y solo si x es true
OR condicional	x || y	Evalúa y solo si x es false
Uso combinado de NULL	x ?? s	Se evalúa como y si x es NULL; de lo contrario, se evalúa como x
Condicional	x ? y : z	Se evalúa como y si x es true y como z si x es false

Figura 3.2: Operadores lógicos, condicionales y NULL

Operadores de igualdad

Expresión	Description
x == y	Igual
x != y	No igual

Figura 3.3: Operadores de igualdad

Operadores relacionales y de tipo

Expresión	Description
x < y	Menor que
x > y	Mayor que
x <= y	Menor o igual que
x >= y	Mayor o igual que
x is T	Devuelve true si x es T; de lo contrario, false
x as T	Devuelve x escrito como T, o NULL si x no es T

Figura 3.4: Operadores relacionales y de tipo

indistintamente una u otro.

Para aplicaciones grandes que manejan un gran volumen de datos es necesario optimizar el espacio que ocupan esos datos, ajustando lo máximo posible el tipo de las variables a los posibles valores que éstas vayan a almacenar.

3.2. Tipos de referencia

Hay dos clases de tipos en C#: tipos de referencia y tipos de valor. Las variables de tipos de referencia almacenan referencias en sus datos (objetos), mientras que las variables de tipos de valor contienen directamente los datos. Con los tipos de referencia, dos variables pueden hacer referencia al mismo objeto y, por lo tanto, las operaciones en una variable pueden afectar 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 .
Las palabras clave siguientes se usan para declarar tipos de referencia:
class: Palabra reservada para crear clases
Interface: Una interfaz contiene solo las firmas de métodos, propiedades, eventos o indicadores.
delegate: La declaración de un tipo delegado es similar a una firma de método. Tiene un valor devuelto y un número cualquiera de parámetros de cualquier tipo.

3.3. Sentencias condicionales

Sentencia if
La sentencia **if** permite elegir entre dos alternativas en la función del valor(verdadero o falso) de cierta condición. Si la condición es verdadera, entonces se ejecuta un fragmento de código, y si es falsa, entonces se ejecuta otro distinto (usando la palabra else e indicando otro bloque de código), o no se ejecuta nada (No indicando el bloque de código else). **Sintaxis de la instrucción if**.
La instrucción **if** debe utilizarse de acuerdo a la siguiente sintaxis:

```
1 if(<Condicion>)
2   <instruccion_if>
3 else
4   <instruccion_else>
```

Ejemplo 3.1

```
using System;
class Ejemplo_3.1 {
  static void Main() {
    string nombre;  //Esta variable de tipo string guarda el
    nombre del usuario
    Console.WriteLine("Escribe tu nombre");
    nombre=Console.ReadLine();
    if(nombre == "Armando")
      Console.WriteLine("Bienvenido Armando!");
    else
      Console.WriteLine("Usuario no valido");

    Console.ReadKey();  //Esta linea es util al ejecturar
    nuestro programa en Visua Studio para evitar que se cierre la
     consola inesperadamente.
  }
}
```

A menudo nos interesa introducir más de una línea de código en nuestros bloques **if**, **else** para esto utilizamos llaves en cada bloque.

```
if(<Condicion>) {
  <instruccion1_if>
  <instruccion2_if>
  <instruccion3_if>
  <instruccion4_if>
}
else {
  <instruccion1_else>
  <instruccion2_else>
  <instruccion3_else>
}
```

En el ejemplo 2 la sentencia *nombre == "Armando"* regresa un valor y ese valor es **true** o **false**

La instrucción switch

En ocaciones hay que tomar un gran número e decisiones dependiendo del valor que tiene una determinada expresión. Esto obliga a utilizar una colección de instrucciones **if** anidadas, tales que todas ellas realizan una comprobación sobre la misma expresión.

Sintaxis de la instrucción switch
La instrucción **switch** debe utilizarse de acuerdo a la siguiente sintaxis:

```
switch (<expresion>) {
  case <valor1>:
    <bloque_de_instrucciones_1>
    break;
  case <valor2>:
    <bloque_de_instrucciones_2>
    break;
  .....
  case <valorn>:
    <bloque_de_instrucciones_n>
    break;
  default:
    <bloque_de_instrucciones>
    break;
}
```

El significado de esta instrucción es la siguiente: se evalúa *expresion*. Si su valor es *valor1* se ejecuta *bloque de instrucciones 1*, si es *valor2* se ejecuta *bloque de instruciones 2*, y así para el resto de valores especificados. Si no es igual a ninguno de esos valores y se incluye la rama **default**, se ejecuta *¡bloque de instrucciones¿*; si no se incluye se pasa directamente a ejecutar la instrucción siguiente al **switch**.
Los valores indicados en cada rama del **switch** han de ser expresiones constantes que produzcan valores de algún tipo básico. No puede haber más de una rama con el mismo valor. Cada bloque de instrucciones de cada rama debe terminar con una instrucción **break** para indicarle que continúe la ejecución con la siguiente instrucción al **switch**.
Ejemplo 3.2.

```
using System;
class Ejemplo_Switch
{
    static void Main()
    {
        Console.WriteLine("Cafes: 1=Chico 2=Mediano 3=Grande");
        Console.Write("Introduzca el cafe deseado: ");
        string s = Console.ReadLine();
        int n = int.Parse(s); //Esta linea hace un casteo de
   cadena a un valor entero
        int cost = 0;
        switch(n)
        {
```

```
13          case 1:
14              cost += 25;
15              break;
16          case 2:
17              cost += 50;
18              break;
19          case 3:
20              cost += 75;
21              break;
22          default:
23              Console.WriteLine("Seleccion invalida, Seleccione
    solo 1, 2, o 3.");
24              break;
25          }
26          if (cost != 0)
27          {
28              Console.WriteLine("Introduzca {0} pesos.", cost);
29          }
30          Console.WriteLine("Gracias por su compra.");
31      }
32  }
```

3.4. Ciclos de repetición

La instrucción while

Permite ejecutar un bloque de instrucciones mientras se cumpla una cierta condición: si la condición es verdadera, entonces se ejecuta el fragmento de código incluido dentro del **while**, y si es falsa, se salta el bucle y no se ejecuta nada.

Sintaxis de la instrucción while

La instrucción while debe utilizarse de acuerdo a la siguiente sintaxis.

```
1   while(<condicion>)
2     <instrucciones>
```

Ejemplo 3.3.

```
1  using System;
2  class Ejemplo_While
3  {
4      static void Main() {
5        int n = 0;
6      while (n < 5) {
```

```
7           Console.WriteLine(n);
8           n++;
9       }
10      }
11  }
```

Este ejemplo imprime los números del 0 al 4, es decir, el código se repite hasta que n sea menor a 5.

El bucle for

Un bucle for ejecuta un conjunto de declaraciones un número específico de veces y tiene la sintaxis.

```
1 for(init; condicion; incremento;) {
2   <Instrucciones>
3 }
```

Un contador es declarado una vez en **init**. A continuación, la **condicion** evalúa el valor del contador y el cuerpo del bucle es ejecutado si la condición es verdadera.

Después de la ejecución del bucle, la declaración de **incremento** actualiza el contador, también llamado la variable de control del bucle.

La condición es evaluada una vez más, y el cuerpo del bucle se repite, sólo deteniéndose cuando la condición se vuelve **falsa**.

Ejemplo 3.4

```
1 using System;
2 class Ejemplo_for {
3   static void Main() {
4     for(int x = 10; x < 15; x++) {
5       Console.WriteLine("El valor de x es: " + x);
6     }
7   }
8 }
```

El anterior ejemplo imprime los números del 10 al 14

En la última sección del **for** puede ir en lugar de x++ x+=3 o x-=2 dependiendo del interés del programador.

Las declaraciones **init** e **incremento** pueden ser omitidas, si no se requieren, pero recuerda que los puntos y comas son obligatorios.

Ejemplo 3.5

```
using System;
class Ejemplo_for {
  static void Main() {
  int x = 10;
    for(; x < 15; ) {
      Console.WriteLine("El valor de x es: " + x);
      x-=3;
    }
  }
}
```

El ciclo **for(;;)** es un bucle infinito.

El Bucle do-while Un bucle do-while es similar a un bucle **while**, excepto que un bucle **do-while** está garantizado a ser ejecutado al menos una vez.

Ejemplo 3.6

```
using System;
class Ejemplo_do_while {
  static void Main() {
  int x = 0;
    do {
      Console.WriteLine("El valor de x es: " + x);
      x++;
    }while(x < 5);
  }
}
```

Es muy importante colocar el **punto y coma** al final de la condición del **while**. si la condición del bucle **do-while** evalúa a **falso** las declaraciones en el **do** aún serán ejecutadas una vez. El bucle **do-while** ejecuta las declaraciones al menos una vez, luego valida la condición. El bucle **while** ejecuta la declaración sólo después de validar la condición.

Ejemplo 3.7

```
using System;
class Ejemplo_do_while {
  static void Main() {
  int x = 42;
    do {
      Console.WriteLine("El valor de x es: " + x);
```

```
7         x++;
8       }while(x < 10);
9     }
10 }
```

El ejemplo anterior imprime "El valor de x es: 42.ª pesar de que la condición del while no se cumpla.

Uso de break

Hemos visto el uso de **break** en la declaración **switch**.
Otro uso de **break** es en los bucles: cuando la declaración **break** es encontrada. Dentro de un bucle, el bucle es terminado inmediatamente y la ejecución del programa es trasladada a la siguiente declaración que sigue al cuerpo del bucle.

Ejemplo 3.8

```
1  using System;
2  class Ejemplo_while {
3    static void Main() {
4    int x = 0;
5      while (x < 20) {
6        if(x == 5)
7          break;
8        Console.WriteLine("El valor de x es: " + x);
9        x++;
10     }
11   }
12 }
```

El ejemplo anterior imprime los números del 0 al 4, si quitamos el break los imprimiría hasta el 19.
Si estás utilizando bucles anidados (Un bucle dentro de otro), la declaración **break** detendrá la ejecución del bucle más interno y comenzará a ejecutar la siguiente línea de código después del bloque.

La declaración continue

La declaración **continue** es similar a la declaración **break**, pero en lugar de finalizar el bucle completamente, salta la iteración actual del bucle y continúa con la siguiente iteración.

Ejemplo 3.9

```
1  using System;
2  class Ejemplo_while {
3    static void Main() {
4      for(int i = 0; i < 10; i++) {
5        if(i == 5)
6          continue;
7        Console.Writeline(i);
8      }
9    }
10 }
```

El anterior ejemplo imprime los números del cero al nueve excepto el 5 ya que la declaración **continue** salta las declaraciones siguientes de esa iteración del bucle.

Capítulo 4

Clases y Objetos

4.1. Conceptos básicos de POO

4.2. Creación de clases

4.3. Constructores

4.4. Propiedades

4.5. Atributos y métodos de instancia

4.6. Miembros estáticos

4.7. Estructuras

4.8. Tipos de referencia vs Tipos de valor

4.9. Clases estáticas y métodos de acceso

Capítulo 5

Control de Acceso

5.1. Namespaces

5.2. Encapsulamiento y modificadores de acceso

5.3. Métodos accesores vs propiedades

Capítulo 6

Arreglos

6.1. Sintaxis y uso de arreglos

6.2. Arreglos multidimensionales

6.3. Clase Array

Capítulo 7

Objetos y métodos

Capítulo 8

Polimorfismo

8.1. Concepto de polimorfismo

8.2. Interfaces y su implementación

8.3. Relación de subtipos y supertipos

Capítulo 9

Herencia

9.1. Herencia (is-a relationship)

9.2. Métodos virtuales

9.3. Palabras reservadas virtual y override

9.4. Clases abstractas y clases selladas

9.5. Clase Object

Parte II

C# Intermedio

Capítulo 10

Excepciones

10.1. Clase Exception

10.2. Bloque try-catch-finally

10.3. Definición de una excepción

10.4. Relanzar Excepciones

Capítulo 11

Strings

11.1. String vs StringBuilder

11.2. Formato de una cadena

Capítulo 12

Manejo de archivos

12.1. Archivos y flujos

12.2. Manejo de sistema de archivos

12.3. Clases File, FileInfo, Directory, DirectoryInfo

12.4. Lectura y escritura de archivos

Capítulo 13

Genéricos

13.1. Necesidad de tipos genéricos

13.2. Métodos genéricos

13.3. Clases genéricas

Capítulo 14

Colecciones

14.1. Listas y Diccionarios

Capítulo 15

Concurrencia

15.1. Clases Thread y Parallel

15.2. Tasks

15.3. Sincronización

Capítulo 16

Lambdas, Delegados y Eventos

16.1. Expresiones Lambda

16.2. Introducción a delegados y eventos

16.3. Creación y uso de delegados

16.4. Multicast delegate

16.5. Uso de eventos

16.6. Clase EventArg

Capítulo 17

LINQ (Checar Entity Framework)

17.1. Introducción a LINQ

17.2. Query syntax

17.3. Métodos de extensión

17.4. Operaciones estándar de consulta

Parte III

C# Avanzado

Capítulo 18

Interfaces gráficas de usuario con Windows Forms

18.1. Introducción a las GUIs

18.2. Manejo básico de eventos

18.3. Propiedades de los controles y Layouts

Capítulo 19

Controles de Windows Forms

Capítulo 20

Introducción a Programación Asíncrona

20.1. Métodos asíncronos

20.2. Palabras async y await

Capítulo 21

WPF (Windows Presentation Foundation)

21.1. ¿Qué es WPF?

21.2. Diferencias entre WPF y Windows Forms

21.3. Mi primera aplicación con WPF

Capítulo 22

Bases de datos con LINQ

22.1. Introducción a las bases de datos relacionales

22.2. LINQ to Entities y ADO.NET

22.3. Operaciones CRUD

Capítulo 23

Control de versiones con Team Explorer y Git

23.1. Configuración de Git y Team explorer

23.2. Manejo de ramas

23.3. Commit