Universidad Nacional Autónoma de México
Programa de Tecnología 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).

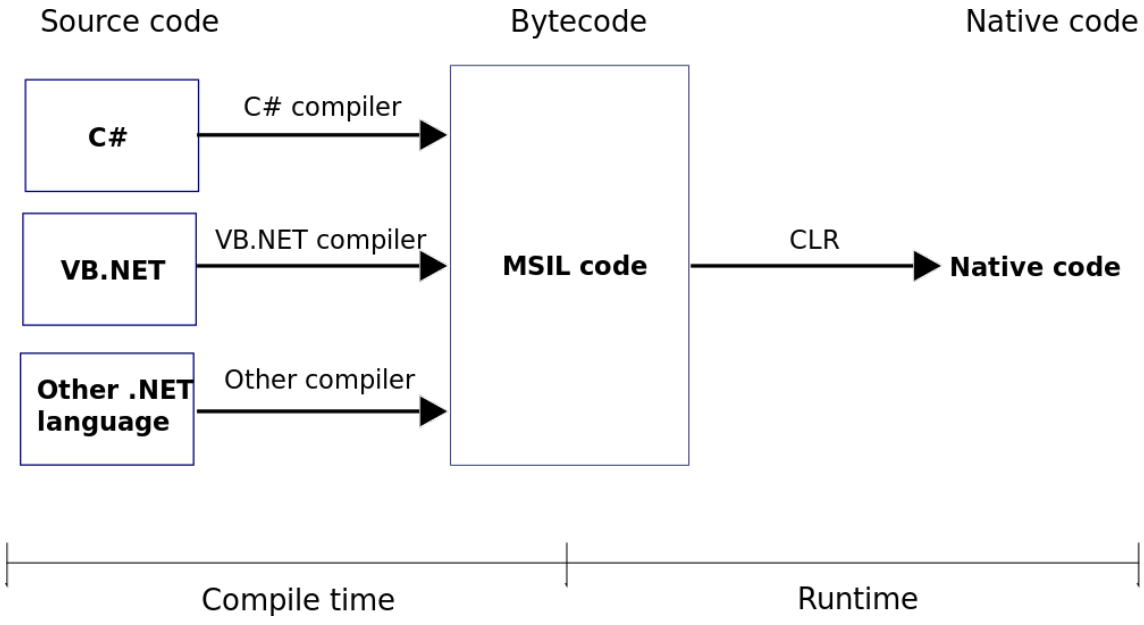

Figura 1.1: Compilación en C#

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

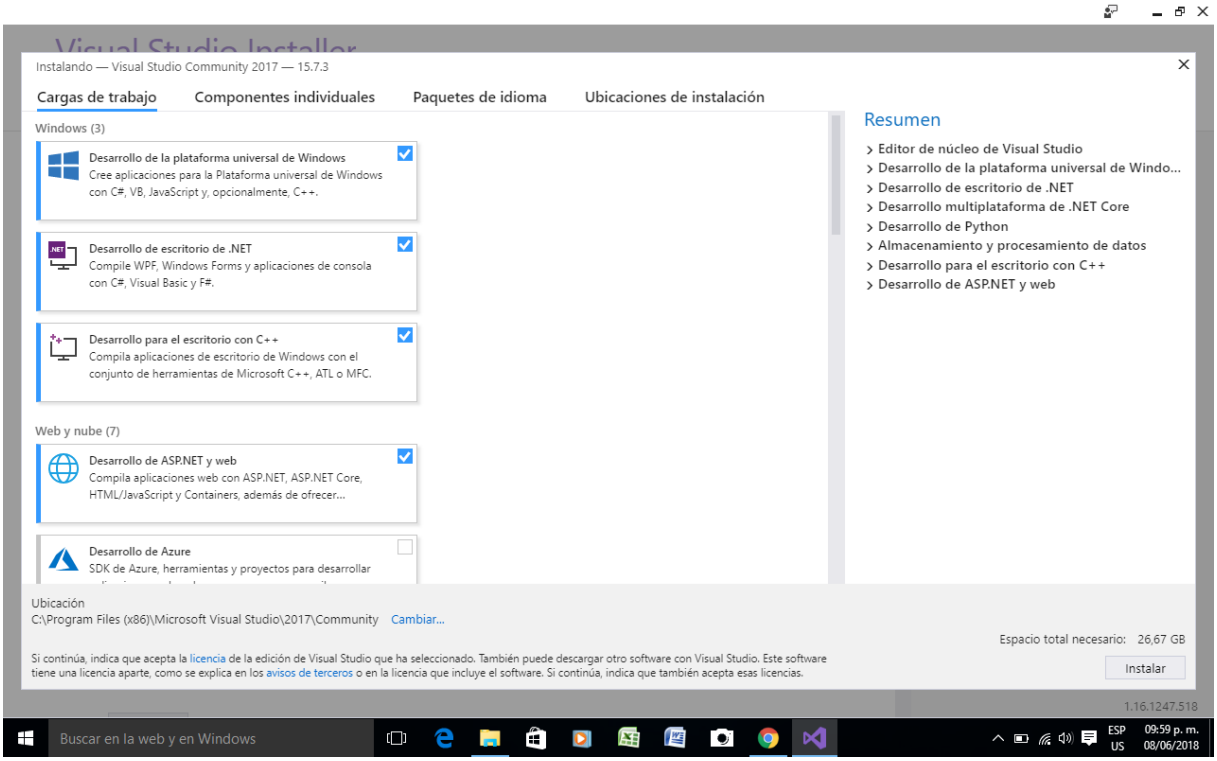
Visual Studio Installer
Instalando — Visual Studio Community 2017 — 15.7.3
Cargas de trabajo
Componentes individuales
Paquetes de idioma
Ubicaciones de instalación
Windows (3)
Desarrollo de la plataforma universal de Windows
Cree aplicaciones para la Plataforma universal de Windows con C#, VB, JavaScript 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

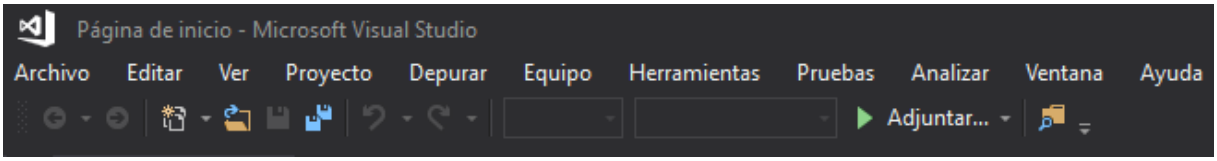

Figura 2.3: Menú superior en Visual Studio

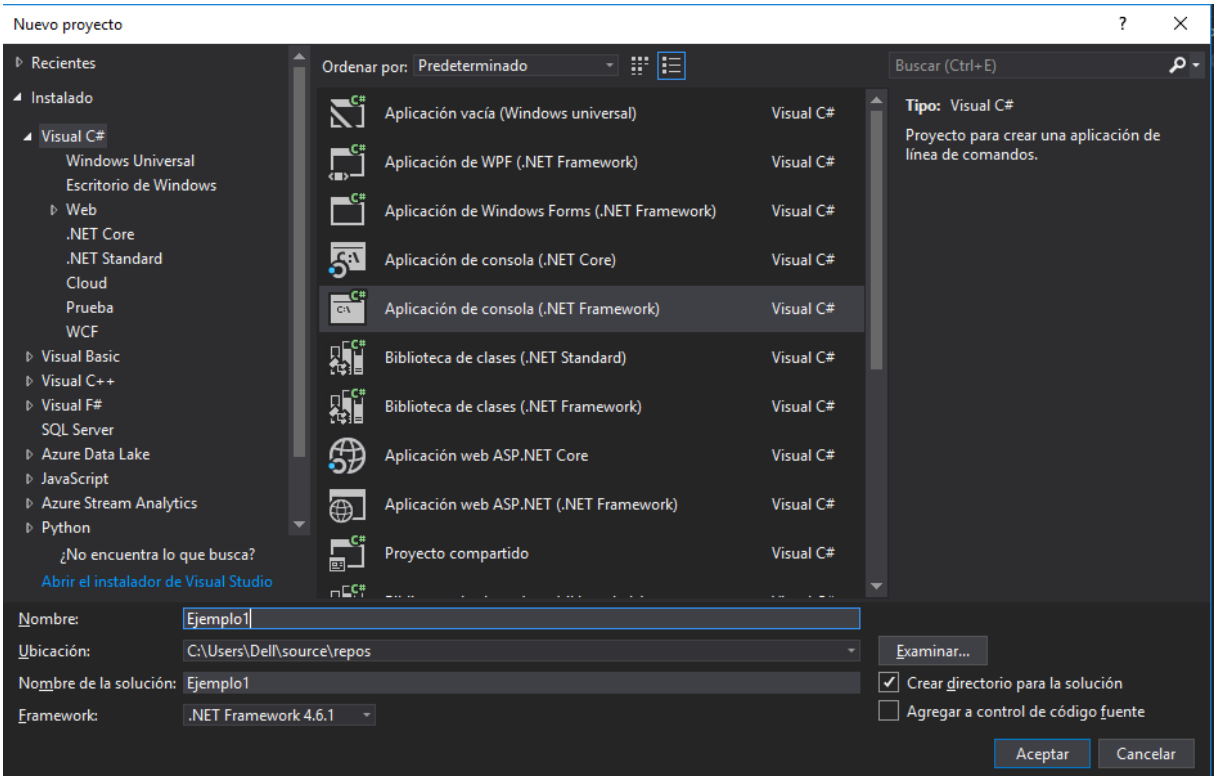

Figura 2.4:

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 *expresión*. 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

C# es un lenguaje que sigue el paradigma orientado a objetos, un paradigma en la programación representa un enfoque particular o filosofía para diseñar soluciones. Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el cómputo. También es importante definir que es una clase y que es un objeto.
Clase: La clase define un tipo de dato para un objeto pero no es un objeto en sí.
Objeto: Un objeto es una entidad concreta basada sobre una clase y es llamado una instancia de una clase.

La programación orientada a objetos está fundamentada sobre los cuatro pilares de la POO que son:
Abstracción: La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿Qué hace?"más que en el "¿Cómo lo hace?"por ejemplo
Encapsulamiento: El encapsulamiento se refiere a restringir el acceso a las funciones internas de una clase.
Beneficios:
- Controlar la manera en que los datos son accedidos o modificados.
- El código es mas flexible y fácil de cambiar a partir de nuevos requerimientos.
- Poder modificar una parte del código sin afectar otras partes del mismo.
Herencia: La herencia permite a la clase derivada reutilizar el código base sin

tener que reescribirlo. Y la clase derivada puede ser personalizada añadiendo mas miembros.De esta manera, la clase derivada extiende la funcionalidad de la clase base.
Polimorfismo: Significa "Tener muchas formas". El polimorfismo es una forma de invocar el mismo método para diferentes objetos y generar diferentes resultados basados en el tipo de objeto.

Estos conceptos los iremos trabajando a mayor profundidad en capítulos posteriores, sin embargo es importante tener una idea de su significado.

Un programa construido mediante un lenguaje orientado a objetos no es más que una colección de objetos que se relacionan entre sí. La forma en que un objeto se comporta y las propiedades que lo definen dependen de la clase a la que pertenece.Una clase puede contener atributos que permiten definir información relativa a la clase y métodos que permiten manipular dichos atributos y definir el comportamiento de los objetos de la clase.

Por ejemplo, mi *coche* es un objeto de la clase *automóvil* y como tal, tiene la capacidad de moverse a una *velocidad constante*, de *acelerarse*, de *detenerse*, etc. Son atributos de mi *coche*, entre otros, su *marca* y *modelo*, el *número de matrícula*, la *velocidad a la que viaja* en un momento dado y la *cantidad de combustible* que lleva en el depósito. Como por ejemplo de métodos tendríamos *arrancar*, *acelerar*, *frenar* y *repostar*. Los métodos definen el modo en que funciona el objeto y pueden modificar ciertos atributos: *repostar*, modificar la *cantidad de combustible* mientras que *acelerar* modifica, además de la *cantidad de combustible*, la *velocidad*.

Cuando se crea un programa con un lenguaje orientado a objetos lo que se definen son las clases de todos los objetos que van a intervenir en el programa. Cuando dicho programa se ejecute, se crearán los objetos a medida que se necesiten, pudiendo haber más de un objeto de cada clase. Por ejemplo, un programa que se encarga de gestionar una biblioteca deberá contener muchos objetos de la clase libro (un objeto por cada ejemplar que existe en la biblioteca) y muchos objetos de la clase lector (uno por cada persona que utiliza dicha biblioteca). Cuando construimos el programa, únicamente creamos una clase libro y una clase lector, Durante el funcionamiento del programa (es decir, en tiempo de ejecución), se irán creando objetos de esas clases a medida que se necesiten, según se den de alta nuevos libros o nuevos usuarios.

4.2. Creación de clases

Una clase es una construcción que permite crear tipos personalizados propios mediante la agrupación de variables de otros tipos, métodos y eventos. Una clase es como un plano. Define los datos y el comportamiento de un tipo. Las clases se declaran mediante la palabra clave **class**.En Visual Studio das click derecho en **Proyecto** y en el submenú seleccionamos la opción **nuevo** y después **class**, automáticamente se crea un nuevo archivo con la estructura básica de una clase.

```
1  public class Coche {
2    //Campos, propiedades, atributos metodos.
3  }
```

4.3. Constructores

Un CONSTRUCTOR de clase en un miembro especial de una clase que es ejecutado cada vez que un nuevo objeto de esa clase es creado.
Un CONSTRUCTOR tiene exactamente el mismo nombre que su clase, es público y no tiene ningún tipo de retorno.
Ejemplo 4.1:

```
1  class Persona {
2    private int edad;
3    public Person() {
4      Console.WriteLine("Hola amigo");
5    }
6  }
```

Ahora, al momento de creación de un objeto del tipo Persona, el constructor es automáticamente invocado. Para hacer una objeto del tipo persona tenemos que especificar el nombre de la clase seguido de un identificador, posteriormente tenemos que hacer uso del operador de igualación y usar la palabra new como en el siguiente ejemplo.
Ejemplo 4.2

```
1 Using System;
2  static void Main(string [] args) {
3  Persona p = new Persona();
4  }
```

La salida del código anterior es “Hola amigo”,

4.4. Propiedades

Una propiedad es un miembro que ofrece un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Las propiedades proporcionan la comodidad de utilizar miembros de datos públicos sin los riesgos que implica el acceso no protegido y sin control ni comprobación a los datos de un objeto. Las propiedades pueden ser utilizadas como di fueran miembros públicos de datos, pero realmente incluyen métodos especiales llamados **descriptores de acceso (accessors)**
El descriptor de acceso de una propiedad contiene las declaraciones ejecutables que ayudan a obtener (leer o computar) o fijar (escribir) un campo correspondiente. Las declaraciones del descriptor de acceso pueden incluir un descriptor de acceso **get**, un descriptor de acceso **set**, o ambas
Ejemplo 4.3

```
1 class Persona {
2   private string nombre;
3   public string Nombre {
4     get {return name; }
5     set { nombre = value }
6   }
7 }
```

La clase Persona tiene una propiedad **Nombre** que tiene tanto el descriptor de acceso **set** como el descriptor de acceso **get**.
El descriptor de acceso set es utilizado para asignar un valor a la variable nombre mientras que get es utilizado para retornar su valor. **value** es una palabra clave especial, la cual representa el valor que asignamos a una propiedad utilizando el descriptor de acceso **set**. Una vez que la propiedad está definida, podemos utilizarla para asignar y leer el miembro privado.
Ejemplo 4.4

```
1 class Persona {
2   private string nombre;
3   public string Nombre {
4     get {return nombre; }
5     set { nombre = value }
6   }
```

```
7  }
8
9  static void Main() {
10   Persona p = new Persona();
11   p.Name = "Armando";
12   Console.WriteLine(p.name);
13 }
```

La propiedad es accedida por su nombre, tal cual cualquier otro miembro público de la clase.
Cualquier descriptor de acceso de una propiedad puede ser omitido. Por ejemplo, el siguiente código crea una propiedad que es sólo lectura:
Ejemplo 4.5

```
1 class Persona {
2   private string nombre;
3   public string Nombre {
4     get {return nombre; }
5   }
6 }
```

Una propiedad puede también ser **privada**, por lo que sólo podrá ser invocada desde dentro de la clase.

La utilidad de las propiedades es que tenemos la opción de controlar la lógica de acceso a la variable. Por ejemplo, puedes validar si el valor de **edad** es mayor que 0, antes de asignarlo a la variable:
4.6

```
1  class Persona {
2    private string edad;
3    public string Edad {
4      get {return edad; }
5      set {
6        if(value > 0)
7        edad = value
8      }
9    }
10 }
```

Cuando no necesitamos ninguna lógica personalizada, C# provee un mecanismo rápido y efectivo para declarar miembros privados a través de sus propiedades.

Por ejemplo, para crear un miembro privado que sólo pueda ser accedido a través de los descriptores de acceso **get** y **set** de la propiedad nombre, utiliza la siguiente sintaxis.

```
public string Nombre {get; set;}
```

Como puedes ver, no es necesario declarar el campo privado "nombre"por separado.

4.7

```
class Persona {
  public string Nombre { get; set;}
  }
}

static void Main() {
  Persona p = new Persona();
  p.Name = "Armando";
  Console.WriteLine(p.name);
}
```

La salida del código anterior es "Armando"

4.5. Atributos y métodos de instancia

Los atributos son todas aquellas características que le asociamos a un objeto de una clase definida.
Los métodos representan todas aquellas acciones que puede realizar o se pueden llevar a cabo sobre un objeto de una clase.
Los métodos de instancia son aplicables a una instancia de la clase en particular.
También mediante el método constructor podemos pedir al usuario que indique la edad de la persona y esta será asignada al momento de hacer la instancia.

Ejemplo 4.8

```
class Persona {
  private int edad;
  public Person(int edad) {
    this.edad = edad;
    Console.WriteLine("Hola amigo, tienes la edad de " + edad);
  }
}
```

```
8
9  static void Main(string [] args) {
10 Persona p = new Persona(18);
11 }
12 /*Salida: Hola amigo, tienes la edad de 18*/
```

En el ejemplo anterior la palabra **this** hace referencia a la misma clase, es decir el parámetro recibido, en este caso 18, lo va a asignar al atributo edad.

4.6. Miembros estáticos

Un miembro estático es aquel que pertenece a la propia clase en vez de a un objeto específico. El modificador static se utiliza para declarar los miembros de las clases(variables, métodos, propiedades) como estáticos.

Ejemplo 4.9

```
1 class Empleado {
2   public static int contador = 0;
3   public Empledo() {
4     contador++;
5   }
6 }
```

En este caso hemos declarado una variable miembro pública **contador**, la cual es **estática**. El constructor de la clase incrementa la variable **count** en uno.

Sin importar cuantos objetos **Empleado** sean instanciados, siempre habrá una sola variable **count** que pertenece a la clase **Empleado** porque fue declarada **estática**.

4.7. Estructuras

Una estructura es un tipo de valor que normalmente se usa para encapsular pequeños grupos de variables relacionadas. Las estructuras también pueden contener constructores, constantes, campos, métodos, propiedades, eventos y tipos anidados.

Ejemplo 4.10

```
1 public struct Libro {
2   public double precio;
3   public string titulo;
```

```
4     public string autor;
5   }
6
7   static void Main() {
8     Libro l;
9     l.titulo = "El retrato de Dorian Gray"
10    l.precio = 100.00;
11    l.autor = "Oscar Wilde"
12
13    Console.WriteLine(l.titulo);
14    /*Salida: El retrato de Dorian Gray*/
15  }
```

Las estructuras no pueden heredar como las clases. Las clases se usan para modelar un comportamiento complejo, las estructuras tienen la intención principal de ser un conjunto simple de variables. Una clase es un tipo de referencia, un struct es un tipo de valor. Los campos no se puede inicializar a menos que sean constantes o estáticos. Una estructura puede implementar interfaces.

4.8. Tipos de referencia vs Tipos de valor

Un tipo de valor almacena su contenido en la memoria asignada en la pila. Cuando una variable de tipo de valor queda fuera de ámbito, porque en el método en que se definió ha finalizado la ejecución, el valor se descarta de la pila. Por esto, es difícil compartir tipos de valor entre clases.

C# tiene dos formas de almacenar datos: por **referencia** y **por valor**.
Los tipos de datos incorporados, como *int* y *double*, se utilizan para declarar variables que son tipos de **valor**. Su valor se almacena en la memoria en una ubicación llamada la **stack**. Por ejemplo, la instrucción de declaración y asignación int x = 10; puede ser pensada como:
El valor de la variable x está ahora almacenada en la **stack**

Los tipos de referencia se usan para almacenar objetos. Por ejemplo, cuando crea un objeto de una clase, se almacena como un tipo de referencia. Los tipos de referencia se almacenan en una parte de la memoria llamada Heap. Cuando crea una instancia de un objeto, los datos para ese objeto se almacenan en el montón, mientras que su dirección de memoria de montón se almacena en la pila. Es por eso que se llama un tipo de referencia: contiene una referencia (la dirección de la memoria) al objeto real en el heap.

Figura 4.1: Representación de la stack

Como puede ver en la figura 4.2, el objeto **p1** de tipo Persona en la stack almacena la dirección de memoria del heap donde está almacenado el objeto real.
Stack se utiliza para la asignación de memoria estática, que incluye todos sus tipos de valores, como x.
Heap se utiliza para la asignación de memoria dinámica, que incluye objetos personalizados, que pueden necesitar memoria adicional durante el tiempo de ejecución de su programa.

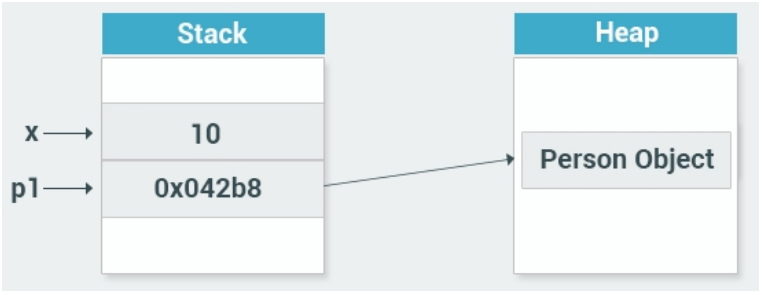

Figura 4.2: Representación de la Heap

4.9. Clases estáticas

Una clase estática es básicamente igual que una clase no estática, pero existe una diferencia: no se pueden crear objetos de una clase estática. El acceso a los miembros de una clase estática se realiza mediante el propio nombre de clase.
Una clase estática es básicamente igual que una clase no estática, pero existe una diferencia: no se pueden crear objetos de una clase estática. El acceso a los miembros de una clase estática se realiza mediante el propio nombre de clase. Las clases estáticas:

Sólo contiene miembros estáticos, no se pueden crear instancias de ella, es de tipo static, no puede contener constructores de instancia.

Ejemplo 4.11

```
1  public static class Convertidor_Temperatura
2  {
3      public static double Celsius_Fahrenheit(string
       temperaturaCelsius)
4      {
5          // Convierte el argumento a double para calcularlo.
6          double celsius = Double.Parse(temperaturaCelsius);
7
8          // Convierte Celsius a Fahrenheit.
9          double fahrenheit = (celsius * 9 / 5) + 32;
10
11         return fahrenheit;
12     }
13
14     public static double Fahrenheit_Celsius(string
       temperaturaFahrenheit)
15     {
16         // Convierte el argumento a double para calcularlo.
17         double fahrenheit = Double.Parse(temperaturaFahrenheit);
18
19         // Convierte Fahrenheit a Celsius.
20         double celsius = (fahrenheit - 32) * 5 / 9;
21
22         return celsius;
23     }
24 }
25
26 class ConvertidorDeTemperatura
27 {
28     static void Main()
29     {
30         Console.WriteLine("Seleccione una opcion para hacer la
       conversion");
31         Console.WriteLine("1. Celsius a Fahrenheit.");
32         Console.WriteLine("2. Fahrenheit a Celsius.");
33         Console.Write(":");
34
35         string opcion = Console.ReadLine();
36         double F, C = 0;
37
38         switch (opcion)
39         {
40             case "1":
41                 Console.Write("Ingrese la temperatura en Celsius
       : ");
```

```
42                  F =Convertidor_Temperatura.Celsius_Fahrenheit(
    Console.ReadLine());
43                  Console.WriteLine("Temperatura en Fahrenheit:
    {0:F2}", F);
44                  break;
45
46              case "2":
47                  Console.Write("Ingrese la temperatura en
    Fahrenheit:  ");
48                  C =Convertidor_Temperatura.FahrenheitToCelsius(
    Console.ReadLine());
49                  Console.WriteLine("Temperatura en Celsius: {0:F2
    }", C);
50                  break;
51
52              default:
53                  Console.WriteLine("Seleccione una opcion valida.
    ");
54                  break;
55          }
56      }
57 }
58 /* Ejemplo:
59   Seleccione una opcion para hacer la conversion
60     1. Celsius a Fahrenheit.
61     2. Fahrenheit a Celsius.
62     :2
63     Ingrese la temperatura en Fahrenheit: 20
64     Temperatura en Celsius: -6.67
65  */
```

Note que en el ejemplo anterior en ningún momento se está haciendo una instancia de la clase *Convertidor_Temperatura* debido a que es una clase estática.

Capítulo 5

Control de Acceso

5.1. Namespaces

El espacio de nombres permite agrupar varias clses que tienen cierta relación lógica entre ellas. Nosotros también podemos definir nuestros propios espacios de nombres, mediante la palabra reservada **namespace**, cada vez que escribimos al comienzo de un programa **using System**, estamos indicando a C# que deseamos tener acceso a las clases definidas en el espacio de nombres **System**, entre las que se encuentran entre otras clases **Console** y **Math**.

Ejemplo 5.1

```
1  namespace Figuras {
2    class Punto {
3      // Atributos: coordenadas del punto en el plano
4      private double x, y;
5      //Constructor
6      public Punto(double x, double y) {
7        this.x = x;
8        this.y = y;
9      }
10     //Propiedades
11     public double X {
12       get {return x;}
13       set { x = value;}
14     }
15     public double Y {
16       get {return y;}
17       set { x = y;}
18     }
19     //Suma a este punto las coordenadas del punto p
20     public void suma(Punto p) {
21       x += p.x;
```

```
22          y += p.y;
23        }
24    } //Fin de la clase
25 } //Fin del espacio de nombres
```

La clase PUNTO representa un punto situado sobre el plano. En la implementación hemos decidido que forme parte de un espacio de nombres FIGURAS. El método SUMA recibe un objeto de tipo PUNTO como argumento e incrementa las coordenadas del punto actual con las del argumento. El siguiente ejemplo muestra el uso de la clase anterior en un programa principal, para esto necesitamos utilizar el espacio de nombres FIGURAS.

Ejemplo 5.2

```
1 using Figuras;
2 using System;
3 class Ejemplo5_2 {
4   public static void Main() {
5     Punto p = new Punto(1.0, 1.0);
6     p.X(3.0);
7     System.Console.WriteLine("x del punto: {0}, y del punto: {1}
      ", p.X, p.Y);
8   }
9 }
```

5.2. Encapsulamiento y modificadores de acceso

Parte del significado de la palabra **encapsulación** es la idea de "rodear.ª una entidad, no solo para mantener lo que está dentro, sino también para protegerlo.
En la programación, la **encapsulación** significa más que simplemente combinar miembros dentro de una clase; también significa restringir el acceso al funcionamiento interno de esa clase.
La encapsulación se implementa mediante el uso de **modificadores de acceso**. Un modificador de acceso define el alcance y la visibilidad de un miembro de la clase.

C# admite los siguientes modificadores de acceso: **public**, **private**, **protected**, **internal**, **protected internal**.

Como se vio en los ejemplos anteriores, el modificador de acceso **public** hace que el miembro sea accesible desde el exterior de la clase.
El modificador de acceso **private** hace que los miembros sean accesibles solo desde dentro de la clase y los oculta desde el exterior.

public: Puede obtener acceso al tipo o miembro cualquier otro código del mismo ensamblado o de otro ensamblado que haga referencia a éste.
private: Solamente puede obtener acceso al tipo o miembro código de la misma clase o struct.
protected: Solamente puede obtener acceso al tipo o miembro el código de la misma clase o struct, o bien de una clase derivada de dicha clase.
internal: Puede obtener acceso al tipo o miembro cualquier código del mismo ensamblado, pero no de un ensamblado distinto.
protected internal: Puede obtener acceso al tipo o miembro cualquier código del ensamblado en el que se declara, o bien desde una clase derivada de otro ensamblado. El acceso desde otro ensamblado debe realizarse dentro de una declaración de clase derivada de la clase en la que se declara el elemento interno protegido y a través de una instancia del tipo de clase derivada.

5.3. Métodos accesores vs propiedades

Los métodos accesores o métodos **get** y **set** son métodos parecidos a las propiedades y nos sirven para tener un mejor control de nuestros atributos.
Ejemplo 5.3

```
1  namespace Figuras {
2    class Punto {
3      // Atributos: coordenadas del punto en el plano
4      private double x, y;
5      //Constructor
6      public Punto(double x, double y) {
7        this.x = x;
8        this.y = y;
9      }
10     //Metodos accesores
11     public double getX() {
12       return x;
13     }
14     public double getY() {
15       return y;
16     }
17     public double setX(double x) {
18       this.x = x;
19     }
```

```
20      public double setY(double y) {
21        this.y = y;
22      }
23      //Suma a este punto las coordenadas del punto p
24      public void suma(Punto p) {
25        x += p.x;
26        y += p.y;
27      }
28    } //Fin de la clase
29  } //Fin del espacio de nombres
```

El uso es el mismo, sin embargo con las propiedades nos podemos ahorrar lineas de código. El siguiente ejemplo muestra el uso de la clase anterior en una clase principal. **Ejemplo 5.4**

```
1  using Figuras;
2  using System;
3  class Ejemplo5_4 {
4    public static void Main() {
5      Punto p = new Punto(1.0, 1.0);
6      p.setX(3.0);
7      System.Console.WriteLine("x del punto: {0}, y del punto: {1}
       ", p.getX, p.getY);
8    }
9  }
```

Capítulo 6

Arreglos

6.1. Sintaxis y uso de arreglos

C# proporciona numerosas clases integradas para almacenar y manipular datos. Un ejemplo de dicha clase es la clase Array.
Una **matriz**,**array**, **vector** o **arreglo** es una estructura de datos que se usa para almacenar una colección de datos. Puedes pensarlo como una colección de variables del mismo tipo.
Por ejemplo, considere una situación en la que necesite almacenar 100 números. En lugar de declarar 100 variables diferentes, puede declarar una matriz que almacena 100 elementos.
Para declarar una matriz, especifique sus tipos de elementos entre corchetes:
Ejemplo 6.1

```
1 int[] miArreglo;
2 int[ ] miArreglo = new int[5]; /*Esta declaracion declara una
    matriz de enteros.Como las matrices son objetos, necesitamos
    instanciarlos con la nueva palabra new:*/
```

Después de crear la matriz, puede asignar valores a elementos individuales usando el número de índice.
Ejemplo 6.2

```
1 int[] miArreglo = new int[5];
2 miArreglo[0] = 23; //Se agrega el numero 23 en el primer
    elemento del arreglo
```

Note que los arreglos en C# y en muchos otros lenguajes de programación el primer elemento es el índice cero, no uno como usualmente estamos acostum-

brados a contar. Podemos inicializar un arreglo al momento de declararlo, o podemos omitir el tamaño del arreglo en la declaración si lo inicializamos, incluso podemos omitir el operador NEW. Podemos acceder a cada elemento especificando el indice de dicho elemento

Ejemplo 6.3

```
1 double[ ] precios = new double[4] {3.6, 9.8, 6.4, 5.9};
2 double[ ] precios = new double[ ] {3.6, 9.8, 6.4, 5.9};
3 double[ ] reales = {3.6, 9.8, 6.4, 5.9};
4
5 Console.WriteLine(reales[0]); //Imprime 3.6
6 Console.WriteLine(reales[3]); //Imprime 5.9
```

A veces es necesario recorrer los elementos de una matriz, haciendo asignaciones de elementos basadas en ciertos cálculos. Esto se puede hacer fácilmente usando bucles.

Por ejemplo, puede declarar una matriz de 10 enteros y asignarle a cada elemento un valor par con el siguiente ciclo:

Ejemplo 6.4

```
1 int[ ] a = new int[10];
2 for (int k = 0; k < 10; k++) {
3   a[k] = k*2;
4 }
```

También podemos usar un bucle para leer los valores de una matriz.

Por ejemplo, podemos mostrar los contenidos de la matriz que acabamos de crear:

Ejemplo 6.5

```
1 for (int k = 0; k < 10; k++) {
2   Console.WriteLine(a[k]);
3 }
```

6.2. Arreglos multidimensionales

Un Arreglo puede tener múltiples dimensiones, un Arreglo multidimensional es declarado de la siguiente forma.

Ejemplo 6.6

```
tipo[, , ... ,] Nombre_del_Arreglo = new tipo[size1, size2, ...,
    sizeN];
int[,]x = new int[3,4];
```

En la figura 6.1 se ilustra un arreglo de 4x3. También podemos inicializar los

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Figura 6.1: Arreglo de 4x3

arreglos multidimensionales.

Ejemplo 6.7

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

Esto creará una matriz con tres filas y dos columnas. Las llaves anidadas se usan para definir valores para cada fila.
Para acceder a un elemento de la matriz, proporcione ambos índices. Por ejemplo, Numeros [2, 0] devolverá el valor 4, ya que accede a la primera columna de la tercera fila.
Vamos a crear un programa que muestre los valores de la matriz en forma de una tabla.

Ejemplo 6.8

```
for (int k = 0; k < 3; k++) {
  for (int j = 0; j < 2; j++) {
    Console.Write(Numeros[k, j]+" ");
  }
  Console.WriteLine();
}
```

Hemos utilizado dos ciclos for anidados, uno para iterar a través de las filas y otro a través de las columnas.
Console.WriteLine (); La instrucción mueve la salida a una nueva línea después de imprimir una fila.
Las matrices pueden tener cualquier cantidad de dimensiones, pero tenga en cuenta que las matrices con más de tres dimensiones son más difíciles de administrar.

Un *jagged array* es una matriz cuyos elementos son matrices. Entonces, básicamente, es una matriz de matrices. La siguiente es una declaración de una matriz unidimensional que tiene tres elementos, cada uno de los cuales es una matriz de enteros de una sola dimensión:
Ejemplo 6.9

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

Cada dimensión es una arreglo, por lo que también puedes inicializar el arreglo durante la declaración de la siguiente forma:
Ejemplo 6.10

```
int[ ][ ] jaggedArr = new int[ ][ ] {
  new int[ ] {1,8,2,7,9},
  new int[ ] {2,4,6},
  new int[ ] {33,42}
};
```

Puedes acceder elementos individuales del arreglo como de muestra en el ejemplo siguiente:
Ejemplo 6.11

```
int x = jaggedArr[2][1]; //42
\\Accede al segundo elemento del tercer arreglo
```

Un **jagged array** es una matriz de arreglos, por lo que una int [] [] es una matriz de int [], cada una de las cuales puede tener diferentes longitudes y ocupar su propio bloque en la memoria.
Una matriz multidimensional (int [,]) es un bloque único de memoria (esencialmente una matriz). Siempre tiene la misma cantidad de columnas para cada fila.

6.3. Clase Array

La clase Array en C# proporciona varias propiedades y métodos para trabajar con matrices.
Por ejemplo, las propiedades **Length** y**Rank** devuelven el número de elementos y el número de dimensiones de la matriz, respectivamente. Puede acceder a ellos usando la sintaxis de punto, al igual que cualquier miembro de la clase:
Ejemplo 6.12

```
1 int[ ] arr = {2, 4, 7};
2 Console.WriteLine(arr.Length);
3 //Salida: 3
4 Console.WriteLine(arr.Rank);
5 //Salida: 1
```

La propiedad **Length** puede ser útil en ciclos **for** donde se necesita especificar el número de veces que se debe ejecutar el ciclo.
Ejemplo 6.12

```
1 int[ ] arr = {2, 4, 7};
2 for(int k=0; k<arr.Length; k++) {
3   Console.WriteLine(arr[k]);
4 }
```

Hay una serie de métodos disponibles para matrices.
Max devuelve el valor más grande.
Min devuelve el valor más pequeño.
Sum devuelve la suma de todos los elementos.
Ejemplo 6.13

```
1 int[ ] arr = { 2, 4, 7, 1};
2 Console.WriteLine(arr.Max()); //Salida: 7
3 Console.WriteLine(arr.Min()); //Salida: 1
4 Console.WriteLine(arr.Sum()); //Salida: 14
```

Capítulo 7

Objetos y métodos

7.1. Sobrecarga de métodos

La sobrecarga de métodos se produce cuando varios métodos tienen el mismo nombre, pero diferentes parámetros.
Por ejemplo, puede tener un método **Print** que muestre su parámetro en la ventana de la consola:
Ejemplo 7.1

```
1 void Print(int a) {
2   Console.WriteLine("Valor: "+a);
3 }
```

El operador + se usa para concatenar valores. En este caso, el valor de **a** se une al texto "Valor:".
Este método solo acepta un argumento entero.
Sobrecargarlo lo hará disponible para otros tipos, como el doble:
Ejemplo 7.2

```
1 void Print(double a) {
2   Console.WriteLine("Valor: "+a);
3 }
```

Ahora, el mismo nombre de método de impresión funcionará tanto para enteros como para dobles.
Al sobrecargar los métodos, las definiciones de los métodos deben diferir entre sí por los tipos y/o el número de parámetros.
Cuando hay métodos sobrecargados, el MÉTODO llamado se basa en los argumentos. Un ARGUMENTO **entero** llamará a la implementación del método

que acepta un parámetro entero. Un argumento **double** llamará a la implementación que acepta un parámetro doble. Múltiples argumentos llamarán a la implementación que acepta la misma cantidad de argumentos.

Ejemplo 7.3

```
static void Print(int a) {
  Console.WriteLine("Valor: " + a);
}
static void Print(double a) {
  Console.WriteLine("Valor: " + a);
}
static void Print(string label, double a) {
  Console.WriteLine(label + a);
}

static void Main(string[] args) {
  Print(11);
  Print(4.13);
  Print("Promedio: ", 7.57);
}
```

No puede sobrecargar las declaraciones de métodos que difieren solo por tipo de retorno.
La siguiente declaración da como resultado un **error**.
int PrintName (int a)
float PrintName (int b)
double PrintName (int c)

7.2. Comparación de objetos

El problema de las referencias afecta también a la comparación de objetos, observe el siguiente código:

Ejemplo 7.4

```
Punto p1 = new Punto(0,0);
Punto p2 = new Punto(0,0);
//Comparamos
if(p1==p2)
  Console.WriteLine("Iguales");
else
  Console.WriteLine("Diferentes");
```

Este fragmento de código escribe "Diferentes" a pesar de que p1 y p2 representan ambos el mismo punto. La razón es porque las variables contienen referencias a los objetos: P1 tiene una referencia a un objeto representando el punto (0,0) y P2 tiene una referencia a un objeto representando el punto (0,0). El operador == no sirve para conocer si se trata de dos objetos iguales, sino si se trata de dos *referencias* al mismo *objeto*, y en este caso no lo son.

Si queremos comparar el contenido de los objetos y no las referencias debemos escribir un método a propósito. Este método siempre se llama en C# **Equals**, y es una buena idea incluirlo como miembro de todas las clases que definamos. La versión final de la clase PUNTO incluyendo el método **Equals** es la siguiente.

Ejemplo 7.5

```
1  namespace Figuras {
2    class Punto {
3      // Atributos: coordenadas del punto en el plano
4      private double x, y;
5      public Punto(double x, double y) {
6        this.x = x;
7        this.y = y;
8      }
9      //Metodos accesores
10     public double getX() {
11       return x;
12     }
13     public double getY() {
14       return y;
15     }
16     public double setX(double x) {
17       this.x = x;
18     }
19     public double setY(double y) {
20       this.y = y;
21     }
22     //Suma a este punto las coordenadas del punto p
23     public void suma(Punto p) {
24       x += p.x;
25       y += p.y;
26     }
27     public bool Equals(Punto p) {
28       return x == p.getX && y == p.getY;
29     }
30   } //Fin de la clase
31 } //Fin del espacio de nombres
```

Ahora podemos escribir el Ejemplo 7.4 y sí se escribirá el mensaje “Iguales”.

7.3. Tipos anónimos

Son una manera cómoda de encapsular un conjunto de propiedades de sólo lectura en un único objeto, Para crearlos se usa el operador **“new”** con un inicializador, no podemos saber de que tipo será nuestra variable, son datos por valor, es decir se crean en la pila y su referencia se pierde una vez terminado el método o estructura de control que lo haya creado. Son una forma conveniente de encapsular un conjunto de propiedades de solo lectura en un objeto, sin tener que crear una clase, son una instancia que no posee clase: **Ejemplo 7.6**

```
var persona = new {
  Nombre = "Marco",
  Apellido = "Aguilar"
}
```

var significa que estamos dejando al compilador la tarea de asignar un tipo a nuestras variables. **Ejemplo 7.7**

```
// Define el tipo anonimo con new y un inicializador
var anonimo = new {
  Nombre = "LMD01",
  Precio = 1234.5,
  Serie = 2
};

// Mostrando los datos del objeto anonimo
  Console.WriteLine("Datos del objeto anonimo");
  Console.WriteLine(anonimo.Nombre);
  Console.WriteLine(anonimo.Precio);
  Console.WriteLine(anonimo.Serie);
  Console.ReadKey();
```

7.4. Lista de parámetros variables

C# permite enviar a un método un número variable de parámetros mediante la palabra **“params”**. Sólo puede haber un parámetro **params** por cada método.
Ejemplo 7.8

```
namespace ParametrosVariables {
  class Program {
    public static void UseParams(params int[] list) {
      for (int i = 0; i < list.Length; i++) {
        Console.Write(list[i] + " ");
      }
      Console.WriteLine();
    }

    public static void UseParams2(params object[] list) {
      for (int i = 0; i < list.Length; i++) {
        Console.Write(list[i] + " ");
      }
      Console.WriteLine();
    }

    static void Main(string[] args) {
      // Se puede especificar una lista separada por
      //  comas del tipo especificado
      UseParams(1, 2, 3, 4);
      UseParams2(1, 'a', "prueba");

      // params acepta cero o mas argumentos
      UseParams2();

      // Se puede pasar un arreglo, mientras coincida con el
      //  tipo del parametro definido, en este caso int
      int[] myIntArray = { 5, 6, 7, 8, 9 };
      UseParams(myIntArray);

      object[] myObjArray = { 2, 'b', "prueba", "hola" };
      UseParams2(myObjArray);

      // La siguiente linea genera un error de compilacion
      //  porque un arreglo de object's no puede convertirse
      //  en un arreglo de enteros.
      //UseParams(myObjArray);
      // La siguiente llamada no causa error porque el arreglo
      //  de enteros se convierte en el primer y unico elemento
      //  de params
      UseParams2(myIntArray, myIntArray);

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

7.5. Modificadores de parámetros out y ref

El modificador **out** indica que los argumentos se van a pasar por referencia. El modificador **ref**, es similar, sólo que éste requiere que se incialice la variable antes de pasarla.

Ejemplo 7.9

```
namespace OutRef
{
  class Program
  {
    public static void Main(string[] args)
    {
      String cadena = "Pedro", cadena2;

      Console.WriteLine(cadena);

      //Metodo Ref, recibe un argumento inicializado.
      MetodoRef(ref cadena);
      Console.WriteLine(cadena);

      //Metodo Out
      MetodoOut(out cadena2);
      Console.WriteLine(cadena2);

      Console.ReadKey(true);
    }

    public static void MetodoRef(ref String palabra){
      palabra = "Hola mundo";
    }

    public static void MetodoOut(out String frase){
      frase = "Te quiero";
    }
  }
}
```

7.6. Llamada de parámetros con nombre

La llamada de parámetros con nombre quita la necesidad de recordar o buscar el orden de los parámetros de los métodos. Es útil cuando se conocen los nombres de los parámetros:

Ejemplo 7.10

```
namespace ParametrosConNombre
{
  class Program
  {
    public static void Main(string[] args)
    {
      float weight, height, imc;

      //Pedimos estatura y peso al usuario
      Console.WriteLine("Ingresa tu peso en kilos: ");
      weight = float.Parse(Console.ReadLine());

      Console.WriteLine("Ingresa tu estatura en metros: ");
      height = float.Parse(Console.ReadLine());

      //Paso de parametros normal
      imc = IMC(weight, height);
      Console.WriteLine("El indice de masa corporal es: "+imc);

      //Paso de parametros con nombre
      imc = IMC(estatura:height, peso:weight);
      Console.WriteLine("El indice de masa corporal es: "+imc);

      //Lo que nunca se debe hacer
      //imc = IMC(estatura:height, weight);

      Console.ReadKey(true);
    }

    public static float IMC(float peso, float estatura){
      return peso/(estatura * estatura);
    }

  }
}
```

Capítulo 8

Polimorfismo

8.1. Concepto de polimorfismo

La palabra **polimorfismo** significa "tener muchas formas". Normalmente, el polimorfismo ocurre cuando hay una jerarquía de clases y se relacionan a través de la **herencia** de una clase base común.
Polimorfismo significa que una llamada a un método miembro provocará que se ejecute una implementación diferente dependiendo del tipo de objeto que invoca el método.
Simplemente, el polimorfismo significa que un único método puede tener varias implementaciones diferentes.

Existen tres tipos de polimorfismo.
Polimorfismo por herencia: cuando una clase hereda de otra y un objeto de la clase heredera puede ser tratado como un objeto de la clase padre.
Polimorfismos por abstracción: cuando se hereda de una clase abstracta y el objeto creado también puede ser tratado como uno de esta clase abstracta.
Polimorfismo por interface: es la posibilidad que tenemos de implementar una **interface** y obtener un comportamiento en común de las clases que implementan la interfaz.
El tema de Herencia se trata más a fondo en el siguiente capítulo.

8.2. Interfaces y su implementación

Una interfaz contiene las definiciones de un grupo de funciones relacionadas que una **clase** o **struct** pueda implementar.
Mediante las interfaces, puede incluir, por ejemplo, comportamiento de varios

orígenes en una clase. Esa función es importante en C# porque el lenguaje no admite la herencia múltiple de clases.
Una interfaz es una clase completamente abstracta, que contiene solo miembros abstractos. Se declara usando la palabra clave **interface**.
Todos los miembros de la interfaz son abstractos por defecto, por lo que no es necesario utilizar la palabra clave abstract.
Además, todos los miembros de una interfaz son siempre públicos, y no se les pueden aplicar modificadores de acceso.
Es común usar la letra mayúscula I como la letra inicial para un nombre de interfaz.
Las interfaces pueden contener propiedades, métodos, etc. pero no pueden contener campos (variables).
Pero, ¿por qué usar interfaces en lugar de clases abstractas?
Una clase puede heredar de solo una clase base, pero puede implementar múltiples interfaces.
Por lo tanto, al usar interfaces, puede incluir el comportamiento de múltiples fuentes en una clase.
Para implementar múltiples interfaces, use una lista de interfaces separadas por comas al crear la clase: clase A: IShape, IAnimal, etc.

Características de las interfaces.
Una interfaz es como una clase base abstracta. Cualquier clase o struct que implemente la interfaz debe implementar todos sus miembros.
Una interfaz no puede ser instanciada directamente.
Las interfaces pueden contener eventos, métodos, y propiedades.
Una clase o struct puede implementar varias interfaces.

La interfaz sólo contiene métodos vacíos (los agrumentos y los valores de retorno deben ser definidos). Implementa la interfaz y define los métodos.Implementa la interfaz y define los métodos de otra manera.
Ejemplo 8.1

```
1  public interface IPerro {
2    string ladrar();
3    string dormir();
4  }
5  public class Chihuahua : IPerro {
6    public string ladrar() {
7      return "Chihuahua ladrando"
8    }
9    public string dormir() {
10     return "Chihuahua durmiendo"
11   }
```

```
12 }
13
14 public class Bulldog : IPerro {
15   public string ladrar() {
16     return "Bulldog ladrando"
17   }
18   public string dormir() {
19     return "Bulldog durmiendo"
20   }
21 }
```

Los dos puntos en las clases antes de *IPerro* significan que esas clases van a hacer uso de la **interface IPerro** y están implementando sus métodos.

Ejemplo 8.2

```
1  namespace Polimorfismo {
2      public interface IFigura {
3          double CalcularArea();
4      }
5
6      public class Circulo : IFigura {
7          public int Radio { get; set; }
8
9          public double CalcularArea()
10         {
11             return Math.PI * Radio * Radio;
12         }
13
14         public void DatosCirculo()
15         {
16             Console.WriteLine("Radio: {0}", Radio);
17         }
18     }
19
20     public class Rectangulo : IFigura {
21         public int Base { get; set; }
22         public int Altura { get; set; }
23
24         public double CalcularArea()
25         {
26             return Base * Altura;
27         }
28
29         public void DatosRectangulo()
30         {
31             Console.WriteLine("Base: {0}, Altura: {1}", Base,
       Altura);
32         }
```

```
33      }
34
35      public class Triangulo : IFigura {
36          public int Base { get; set; }
37          public int Altura { get; set; }
38
39          public double CalcularArea()
40          {
41              return Base * Altura / 2;
42          }
43
44          public void DatosTriangulo()
45          {
46              Console.WriteLine("Base: {0}, Altura: {1}", Base,
    Altura);
47          }
48      }
49
50      class Program {
51          static void Main(string[] args) {
52              // Upcasting
53              IFigura f1 = new Circulo { Radio = 5 };
54              IFigura f2 = new Triangulo { Base = 10, Altura = 8
    };
55              IFigura f3 = new Rectangulo { Base = 4, Altura = 3
    };
56              IFigura f4 = new Circulo { Radio = 9 };
57
58              IFigura[] figuras = new IFigura[] { f1, f2, f3, f4
    };
59
60              foreach (var figura in figuras) {
61                  Console.WriteLine(figura.CalcularArea());
62              }
63
64              // Dowcasting
65              Circulo c = f1 as Circulo;
66              Console.WriteLine("Datos del circulo");
67              c.DatosCirculo();
68          }
69      }
70  }
```

Capítulo 9

Herencia

9.1. Herencia (is-a relationship)

La **herencia** nos permite definir una clase basada en otra clase. Esto hace que crear y mantener una aplicación sea fácil.
La clase cuyas propiedades son heredadas por otra clase se llama clase **Base**.
La clase que hereda las propiedades se llama clase **derivada**.
Por ejemplo, la clase base ANIMAL se puede usar para derivar clases de GATO y PERRO.
La clase derivada hereda todas las características de la clase base y puede tener sus propias características adicionales.

Ejemplo 9.1

```
1 class Animal {
2   public int Piernas {get; set;}
3   public int Edad {get; set;}
4 }
```

A partir de la clase ANIMAL podemos derivar otras clases como la clase PERRO

Ejemplo 9.2

```
1 class Perro : Animal {
2   public Perro() {
3     Piernas = 4;
4   }
5   public void Ladrar() {
6     Console.Write("Woof");
7   }
8 }
```

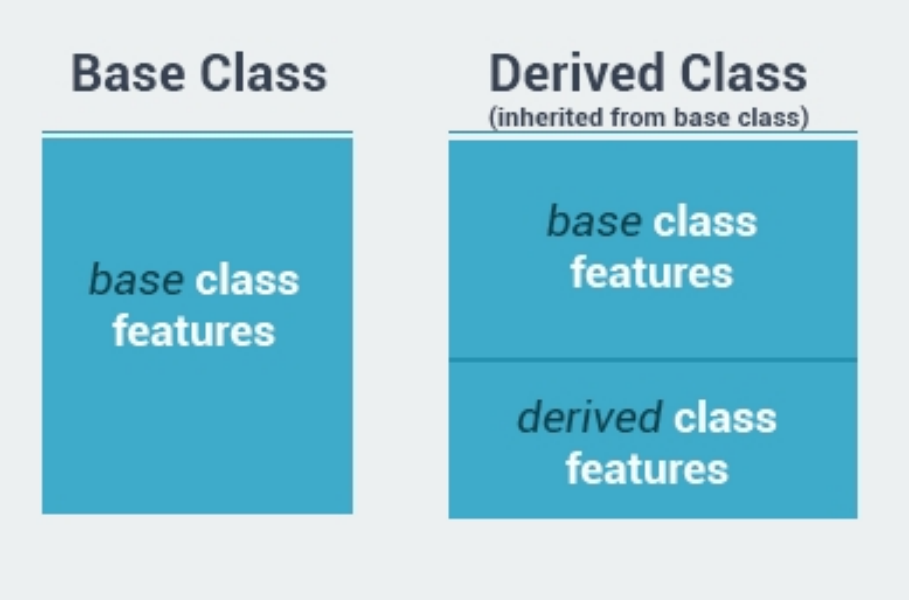

Figura 9.1: Clase base y clase derivada

Tenga en cuenta la sintaxis de una clase derivada. El nombre de la clase derivada, dos puntos y el nombre de la clase base.
Todos los miembros públicos de ANIMAL se convierten en miembros públicos de **Perro**. Es por eso que podemos acceder al miembro de
textitPiernas en el constructor de PERRO.
Ahora podemos instanciar un objeto de tipo PERRO y acceder a los miembros heredados, así como llamar a su propio método *Ladrar*.

Ejemplo 9.3

```
1 static void Main(string[] args) {
2   Perro p = new Perro();
3   Console.WriteLine(p.Piernas);
4   // Salida: 4
5
6   p.Ladrar();
7   //Salida: "Woof"
8 }
```

Una clase base puede tener múltiples clases derivadas, por ejemplo la clase GATO.
La herencia permite que la clase derivada reutilice el código en la clase base sin tener que volver a escribirlo. Y la clase derivada se puede personalizar agregando más miembros. De esta manera, la clase derivada extiende la fun-

cionalidad de la clase base.
Una clase derivada hereda todos los miembros de la clase base, incluidos sus métodos.
Ejemplo 9.4

```
class Persona {
  public void Hablar() {
    Console.WriteLine("Hola amigo");
  }
}
class Estudiante : Persona {
  int numero;
}
static void Main(string[] args) {
  Estudiante e = new Estudiante();
  e.Hablar();
  //Salida: "Hola amigo"
}
```

C# no es compatible con herencia múltiple, por lo que no puede heredar de múltiples clases.
Sin embargo, puede usar interfaces para implementar herencia múltiple.

9.2. Métodos virtuales

Cuando una clase hereda de otra es posible usar los métodos de la clase padre en la clase hija, pero en algunas ocasiones necesitaremos que ese método sea distinto en alguna clase hija, par eso definimos de nuevo el método pero con la palabra reservada **"virtual"**. Para que esto se pueda hacer, la clase padre debe tener definido el método como **virtual** y la clase hija lo tiene que tener definido como **override**.
Ejemplo 9.5

```
public class Figura
    {
        public const double PI = Math.PI;
        protected double x, y;
        public Figura() {
        }
        public Figura(double x, double y) {
            this.x = x;
            this.y = y;
        }

```

```
12          public virtual double Area() {
13              return x * y;
14          }
15      }
```

El modificador de acceso **protected** indica que las variables va a poder ser utilizadas en la clase base y en las hijas. El método **Area** está declarado como **virtual** para que en una clase hija lo podamos sobrescribir, es decir podamos cambiar la implementación.

Ejemplo 9.6

```
1  public class Circulo : Figura {
2    public Circulo(double r) : base(r, 0) {
3    }
4
5    public override double Area() {
6      return PI * x * x;
7    }
8  }
9
10 static void Main() {
11   double r = 3.0;
12   Figura c = new Circle(r);
13
14   Console.WriteLine("Area del Circulo = {0:F2}", c.Area());
15
16   //Salida: 28.27
17 }
```

9.3. Clases abstractas y clases selladas

El polimorfismo se usa cuando tiene diferentes clases derivadas con el mismo método, que tiene implementaciones diferentes en cada clase. Este comportamiento se logra a través de métodos virtuales que se anulan en las clases derivadas.

En algunas situaciones, no existe una necesidad significativa de que el método virtual tenga una definición separada en la clase base.

Estos métodos se definen con la palabra clave **abstract** y especifican que las clases derivadas deben definir ese método por sí mismas.

No puede crear objetos de una clase que contenga un método abstracto, por lo que la clase en sí misma debe ser abstracta.

Podríamos usar un método abstracto en la clase FIGURA:
Ejemplo 9.7

```
abstract class Figura {
   public abstract void Dibujar();
}
```

Como puede ver, el método *Dibujar* es abstracto y, por lo tanto, no tiene cuerpo. Ni siquiera necesitas las llaves; acaba la declaración con un punto y coma.
La clase Figura en sí misma debe declararse abstracta porque contiene un método abstracto. Las declaraciones de método abstracto solo se permiten en clases abstractas.
Recuerde, las declaraciones de métodos abstractos solo se permiten en clases abstractas. Los miembros marcados como abstractos o incluidos en una clase abstracta deben implementarse por clases que se derivan de la clase abstracta. Una clase abstracta puede tener múltiples miembros abstractos.

Una clase abstracta está destinada a ser una clase base de otras clases. Actúa como una plantilla para sus clases derivadas.
Ahora, teniendo la clase abstracta, podemos derivar las otras clases y definir sus propios métodos *Dibujar*. **Ejemplo 9.8**

```
abstract class Figura {
  public abstract void Dibujar();
}
class Circulo : Figura {
  public override void Dibujar() {
    Console.WriteLine("Dibujando Cirulo");
  }
}
class Rectangulo : Figura {
  public override void Dibujar() {
    Console.WriteLine("Dibujando Rectangulo");
  }
}
static void Main(string[] args) {
  Figura c = new Circulo();
  c.Dibujar();
  //Salida: "Dibujando circulo"
}
```

Las clases abstractas tienen las siguientes características:
•Una clase abstracta no puede ser instanciada.

•Una clase abstracta puede contener métodos abstractos.
•Una clase no abstracta derivada de una clase abstracta debe incluir implementaciones reales de todos los métodos abstractos heredados.

Clases Selladas

Una clase puede evitar que otras clases la hereden, o cualquiera de sus miembros, mediante el uso del modificador **sealed**.
Ejemplo 9.9

```
1 sealed class Animal {
2   //Codigo
3 }
4 class Perro : Animal { } //Error
```

En este caso, no podemos derivar la clase Perro de la clase Animal porque Animal está sellado. La palabra reservada **sealed** proporciona un nivel de protección a su clase para que otras clases no puedan heredar de ella.
No es posible modificar una clase abstracta con el modificador **sealed** porque los dos modificadores tienen significados opuestos. El modificador sealed impide que una clase se herede y el modificador **abstract** requiere que se herede una clase.

9.4. Clase Object

object es el alias de la clase **Object**, esta clase es la clase base de todas en la jerarquía de herencia de .Net.
La clase **Object** solo cuenta con un constructor, el cual no recibe argumentos.

Métodos

	Nombre	Descripción
	Equals(Object)	Determina si el objeto especificado es igual al objeto actual.
S	Equals(Object, Object)	Determina si las instancias del objeto especificado se consideran iguales.
	Finalize()	Permite que un objeto intente liberar recursos y realizar otras operaciones de limpieza antes de que sea reclamado por la recolección de elementos no utilizados.
	GetHashCode()	Sirve como la función hash predeterminada.
	GetType()	Obtiene el Type de la instancia actual.
	MemberwiseClone()	Crea una copia superficial del Object actual.
S	ReferenceEquals(Object, Object)	Determina si las instancias de Object especificadas son la misma instancia.
	ToString()	Devuelve una cadena que representa el objeto actual.

Figura 9.2: Tabla de métodos de la clase object

Parte II

C# Intermedio

Capítulo 10

Excepciones

10.1. Definición de una excepción

Una excepción es un problema que ocurre durante la ejecución del programa. Las excepciones causan la terminación anormal del programa.
Una excepción puede ocurrir por muchas razones diferentes. Algunos ejemplos:
•Un usuario ha ingresado datos no válidos.
•No se puede encontrar un archivo que debe abrirse.
•Se perdió una conexión de red en medio de las comunicaciones.
• Memoria insuficiente y otros problemas relacionados con los recursos físicos.
Por ejemplo, el siguiente código producirá una excepción cuando se ejecute porque solicitamos un índice que no existe:
Ejemplo 10.1

```
1 int[] arr = new int[] { 4, 5, 8 };
2 Console.Write(arr[8]);
```

Como puede ver, las excepciones son causadas por error del usuario, error del programador o problemas de recursos físicos. Sin embargo, un programa bien escrito debe manejar todas las excepciones posibles.

10.2. Bloque try-catch-finally

C# proporciona un mecanismo flexible llamado la declaración **try-catch** para manejar excepciones para que un programa no falle cuando se produce

un error.

Ejemplo 10.2

```
try {
  int[] arr = new int[] { 4, 5, 8 };
  Console.Write(arr[8]);
}
catch(Exception e) {
  Console.WriteLine("ha ocurrido un error");
}
//Salida: "ha ocurrido un error"
```

El código que podría generar unaEXCEPCIÓN se coloca en el bloque **try**. Si ocurre una EXCEPCIÓN, los bloques **catch** se ejecutan sin detener el programa.
El tipo de EXCEPCIÓN que desea capturar aparece entre paréntesis después de la palabra reservada **catch**.
Usamos el tipo de Excepción general para manejar todo tipo de excepciones.
También podemos usar el objeto de excepción e para acceder a los detalles de la excepción, como el mensaje de error original (e.Message):

Ejemplo 10.3

```
try {
  int[] arr = new int[] { 4, 5, 8 };
  Console.Write(arr[8]);
}
catch(Exception e) {
  Console.WriteLine(e.Message);
}
//Salida: El indice esta fuera de los limites de la matriz.
```

También puede capturar y manejar diferentes excepciones por separado.

Un único bloque **try** puede contener múltiples bloques **catch** que manejan diferentes excepciones por separado.
El manejo de excepciones es particularmente útil cuando se trata de la entrada del usuario.
Por ejemplo, para un programa que solicita la entrada del usuario de dos números y luego genera su cociente, asegúrese de manejar la división entre cero, en caso de que el usuario ingrese 0 como el segundo número.

Ejemplo 10.4

```
int x, y;
try {
  x = Convert.ToInt32(Console.Read());
  y = Convert.ToInt32(Console.Read());
  Console.WriteLine(x / y);
}
catch (DivideByZeroException e) {
  Console.WriteLine("No es posible dividir entre 0");
}
catch(Exception e) {
  Console.WriteLine("Ha ocurrido un error");
}
```

El código anterior maneja la excepción **DivideByZeroException** por separado. el último **catch** maneja todas las demás excepciones que puedan ocurrir. Si se manejan múltiples excepciones, el tipo de excepción se debe definir al final.
Ahora, si el usuario ingresa 0 para el segundo número, se mostrará "No es posible dividir entre 0".
Si, por ejemplo, el usuario ingresa valores no enteros, se mostrará "Se produjo un error".

Los siguientes tipos de excepciones son algunos de los más utilizados: **FileNotFoundException**, **FormatException**, **IndexOutOfRangeException**, **InvalidOperationException**, **OutOfMemoryException**.

Se puede usar un bloque **finally** opcional después de los bloques **catch**. El bloque **finally** se usa para ejecutar un conjunto dado de declaraciones, ya sea que se genere una excepción o no.**Ejemplo 10.5**

```
int result=0;
int num1 = 8;
int num2 = 4;
try {
  result = num1 / num2;
}
catch (DivideByZeroException e) {
  Console.WriteLine("Error");
}
finally {
  Console.WriteLine(result);
}
```

El bloque **finally** se puede usar, por ejemplo, cuando trabajas con archivos u otros recursos. Deben cerrarse o liberarse en el bloque **finally**, ya sea que se genere una excepción o no.

10.3. Relanzar Excepciones

Además de capturar excepciones también podemos lanzar nuestras propias excepciones. Imaginemos que queremos escribir una clase para representar fracciones a través de su numerador y de su denominador. El constructor recibirá ambos valores, dos enteros, pero el denominador no puede se cero, porque en ese caso la fracción estaría mal definida. Una primera solución podría ser aprovechar una de las excepciones definidas en C#, como **ArgumetException**, para indicar que el argumento pasado al método constructor es incorrecto.

Ejemplo 10.6

```
1  using System;
2  namespace aritmetica {
3    class Fraccion {
4      private int numerador, denominador;
5      public Fraccion(int numerador, int denominador) {
6        if(denominador == 0)
7          throw new ArgumenException("El denominador no puede ser
       cero");
8        this.numerador = numerador;
9        this.denominador = denominador;
10     }
11     //otros metodos
12   } //Fin de la clase Fraccion
13 }
```

El constructor comprueba si el denominador es cero, y en este caso lanza una excepción mediante la palabra clave **throw**, que debe estar seguida por un objeto de tipo **ArgumentException** recibe un **string** como parámetro, valor que se almacenará en la propiedad **Message** del objeto. Este constructor, junto con el constructor por defecto, es el más usado habitualmente para inicializar objetos de tipo excepción.

Si se lanza la excepción, la ejecución del constructor se interrumpe y el resto del código no llega a ejecutarse. El objeto no queda construido y, si el programa que está utilizando la clase no lo impide mediante un bloque **try**, el programa se interrumpe. Es siguiente ejemplo ilustra el uso de esta clase.

Ejemplo 10.7

```
1  using System;
2  using aritmetica;
3
4  class Ejemplo10_7 {
5    public static void Main() {
6      int a, b;
7      try {
8        Console.Write("Numerador: ");
9        a = int.Parse(Console.ReadLine());
10       Console.Write("Denominador: ");
11       b = int.Parse(Console.ReadLine());
12
13       Fraccion f = new Fraccion(a, b);
14     }
15     catch (ArgumetException e) {
16       Console.WriteLine("Error: {0}", e.Message);
17     }
18     carch (FormatException) {
19       Console.WriteLine("Error: Datos no numericos");
20     }
21   }
22 }
```

El programa muestra el mensaje asociado a la excepción si se trata de un objeto de tipo **ArgumentException** o escribe un mensaje directamente si se trata de un error al convertir los datos a tipo entero.

Nótese que al tratar los errores mediante excepciones consiguiendo una gestión más simple de las situaciones derivadas de ese error. En el ejemplo de la fracción, en vez de lanzar una excepción podemos tener otras alternativas: Permitir crear fracciones incorrectas, o añadir un atributo VÁLIDO para indicar si la fracción es o no válida, etc. Estas alternativas ofrecen mayores dificultades en el desarrollo del resto del programa, porque permiten que existan y convivan objetos de tipo fracción "especiales" con objetos normales, requiriendo un tratamiento distinto en la mayoría de métodos que usen fracciones.

Capítulo 11

Strings

11.1. String vs StringBuilder

Es común pensar en cadenas como arreglos de caracteres. En realidad, las cadenas en C# son objetos.
Cuando declaras una variable de cadena, básicamente haces una instancia de un objeto de tipo *String*.
Los objetos *String* admiten una cantidad de propiedades y métodos útiles:

Length: devuelve la longitud de la cadena.
IndexOf (valor) devuelve el índice de la primera aparición del valor dentro de la cadena.
Insert (índice, valor) inserta el valor en la cadena a partir del índice especificado.
Remove (índice) elimina todos los caracteres en la cadena después del índice especificado.
Replace(Valor Antiguo, Valor nuevo) reemplaza el valor especificado en la cadena.
Substring (índice, longitud) devuelve una subcadena de la longitud especificada, comenzando desde el índice especificado. Si no se especifica la longitud, la operación continúa hasta el final de la cadena.
Contains (valor) devuelve verdadero si la cadena contiene el valor especificado.

Ejemplo 11.1

```
1 string a = "Texto";
2 Console.WriteLine(a.Length);
3 //Salida: 5
4
```

```
Console.WriteLine(a.IndexOf('t'));
//Salida: 3

 a = a.Insert(0, "Esto es ");
Console.WriteLine(a);
//Salida: "Esto es Texto"

a = a.Replace("Esto es", "Yo soy");
Console.WriteLine(a);
//Salida: "Yo soy Texto"

if(a.Contains("Texto"))
   Console.WriteLine("encontrado");
//Salida: "encontrado"

a = a.Remove(6);
Console.WriteLine(a);
//Salida: "Yo soy"

a = a.Substring(3);
Console.WriteLine(a);
//Salida: "soy"
```

Vamos a crear un programa que tome una cadena, reemplace todas las ocurrencias de la palabra "dog"por çat.eimprima solo la primera oración.

Ejemplo 11.2

```
string text = "This is some text about a dog. The word dog
    appears in this text a number of times. This is the end.";

text = text.Replace("dog", "cat");
text = text.Substring(0, text.IndexOf(".")+1);

Console.WriteLine(text);
//Salida: "This is some text about a cat."
```

El código anterior reemplaza todas las apariciones de "dog"por çat". Después de eso, toma una subcadena de la cadena original comenzando desde el primer índice hasta la primera aparición de un carácter de punto.
Agregamos uno al índice del período para incluir el período en la subcadena.
C# proporciona una sólida colección de herramientas y métodos para trabajar y manipular cadenas. Podría, por ejemplo, encontrar la cantidad de veces que una palabra específica aparece en un libro con facilidad, usando esos métodos.

textbfString Builder
La clase **System.Text.StringBuilder** se puede usar cuando desee modificar una cadena sin crear un nuevo objeto. Por ejemplo, usar la clase **StringBuilder** puede aumentar el rendimiento al concatenar muchas cadenas juntas en un bucle.
Puedes crear una nueva instancia de la clase **StringBuilder** inicializando su variable con uno de los métodos constructor sobrecargados.
Aunque **StringBuilder** es un objeto dinámico que le permite expandir el número de caracteres en la cadena que encapsula, puede especificar un valor para la cantidad máxima de caracteres que puede contener. Este valor se denomina capacidad del objeto y no debe confundirse con la longitud de la cadena que contiene el actual **StringBuilder**. Por ejemplo, puede crear una nueva instancia de la clase StringBuilder con la cadena "Hola", que tiene una longitud de 5, y puede especificar que el objeto tenga una capacidad máxima de 25. Cuando modifica el **StringBuilder**, no lo hace reasigne el tamaño por sí mismo hasta que se alcance la capacidad. Cuando esto ocurre, el nuevo espacio se asigna automáticamente y la capacidad se duplica.
Ejemplo 11.3

```
1 StringBuilder miCadena1 = new StringBuilder("Hola mundo");
2 StringBuilder miCadena2 = new StringBuilder("Hola mundo", 25);
```

Además, puede usar la propiedad **Capacity** de lectura / escritura para establecer la longitud máxima de su objeto. El siguiente ejemplo usa la propiedad **Capacity** para definir la longitud máxima del objeto.
Ejemplo 11.4

```
1 miCadena1.Capacity = 25;
```

También podemos hacer uso de otros métodos como:
Append: El método Append se puede usar para agregar texto o una representación de cadena de un objeto al final de una cadena representada por **StringBuilder** actual.
Insert: El método **Insert** agrega una cadena u objeto a una posición específica en el objeto **StringBuilder** actual.
textbfRemove: Puede utilizar el método **Remove** para eliminar un número especificado de caracteres del objeto **StringBuilder** actual, comenzando en un índice especificado basado en cero.
textbfReplace: El método **Replace** se puede usar para reemplazar caracteres

dentro del objeto **StringBuilder** con otro carácter especificado.

Ejemplo 11.5

```
1  StringBuilder miCadena = new StringBuilder("Hola mundo!");
2  miCadena.Append(" Que bonito dia!.");
3  Console.WriteLine(miCadena);
4  //Salida: Hola mundo! Que bonito dia!
5
6  StringBuilder miCadena = new StringBuilder("Hola mundo");
7  myStringBuilder.Insert(5,"bonito ");
8  Console.WriteLine(miCadena);
9  //Salida: Hola bonito mundo
10
11
12 StringBuilder miCadena = new StringBuilder("Hola Mundo!");
13 miCadena.Remove(4,7);
14 Console.WriteLine(miCadena);
15 //Salida: Hola
16
17
18 StringBuilder miCadena = new StringBuilder("Hola Mundo!");
19 myStringBuilder.Replace('!', '?');
20 Console.WriteLine(miCadena);
21 //Salida: Hola Mundo?
```

Capítulo 12

Manejo de archivos

12.1. Archivos y flujos

Todos los datos que un programa utiliza durante su ejecución se encuentran en sus variables, que están almacenadas en la memoria RAM del computador.
La memoria RAM es un medio de almacenamiento **volátil**: cuando el programa termina, o cuando el computador se apaga, todos los datos se pierden para siempre.
Para que un programa pueda guardar datos de manera permanente, es necesario utilizar un medio de almacenamiento **persistente**, de los cuales el más importante es el disco duro.
Los datos en el disco duro están organizados en archivos. Un **archivo** es una secuencia de datos almacenados en un medio persistente que están disponibles para ser utilizados por un programa. Todos los archivos tienen un nombre y una ubicación dentro del sistema de archivos del sistema operativo.
Los datos en un archivo siguen estando presentes después de que termina el programa que lo ha creado. Un programa puede guardar sus datos en archivos para usarlos en una ejecución futura, e incluso puede leer datos desde archivos creados por otros programas.
Un programa no puede manipular los datos de un archivo directamente. Para usar un archivo, un programa siempre abre el archivo y lo asigna a una variable, que llamaremos el **archivo lógico**. Todas las operaciones sobre un archivo se realizan a través del archivo lógico.
Dependiendo del contenido, hay muchos tipos de archivos. Nosotros nos preocuparemos sólo de los **archivos de texto**, que son los que contienen texto, y pueden ser abiertos y modificados usando un editor de texto como el Bloc de Notas. Los archivos de texto generalmente tienen un nombre terminado

en .TXT.

12.2. Clases File, FileInfo, Directory, DirectoryInfo

File: Proporciona métodos estáticos para crear, copiar, eliminar, mover y abrir archivos.
FileInfo: Proporciona métodos de instancia para crear, copiar, eliminar, mover y abrir archivos y contribuye a la creación de FileStream. La clase File proporciona métodos estáticos.
Directory: Proporciona métodos estáticos para crear, mover y enumerar archivos en directorios y subdirectorios.
DirectoryInfo: Proporciona métodos de instancia para crear, mover y enumerar archivos en directorios y subdirectorios.

12.3. Lectura y escritura de archivos

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 y más. La clase **File** es uno de ellos.
Ejemplo 12.1

```
1 string str = "Algo de texto";
2 File.WriteAllText("test.txt", str);
```

El método **WriteAllText ()** crea un archivo con la ruta especificada y le escribe el contenido. Si el archivo ya existe, se sobrescribe.
El código anterior crea un archivo test.txt y escribe el contenido de la cadena str en él.
Para usar la clase **File**, necesita usar el espacio de nombres **System.IO: using System.IO;**
Puede leer el contenido de un archivo usando el método **ReadAllText** de la clase **File**.
Ejemplo 12.2

```
1 string txt = File.ReadAllText("test.txt");
2 Console.WriteLine(txt);
```

Esto generará el contenido del archivo test.txt.
Los siguientes métodos están disponibles en la clase **File**:
AppendAllText (): agrega texto al final del archivo.
Create (): Crea un archivo en la ubicación especificada.
Delete (): Borra el archivo especificado.
Exists () Determina si el archivo especificado existe.
Copy (): copia un archivo en una nueva ubicación.
Move () Mueve un archivo especificado a una nueva ubicación
Todos los métodos cierran automáticamente el archivo después de realizar la operación.

Capítulo 13

Genéricos

13.1. Necesidad de tipos genéricos

Los genéricos introducen en **.NET Framework** el concepto de parámetros de tipo, lo que le permite diseñar clases y métodos que aplazan la especificación de uno o varios tipos hasta que el código de cliente declare y cree una instancia de la clase o el método. Por ejemplo, al usar un parámetro de tipo genérico T puede escribir una clase única que otro código de cliente puede usar sin incurrir en el costo o riesgo de conversiones en tiempo de ejecución.
Los genéricos se usan para:
•Maximizar la reutilización del código, la seguridad de tipos y el rendimiento.
•El uso más común de los genéricos es crear clases de colección.
•La biblioteca de clases **.NET Framework** contiene varias clases de colección genéricas nuevas en el espacio de nombres **System.Collections.Generic**. Estas se deberían usar siempre que sea posible en lugar de clases como **ArrayList** en el espacio de nombres **System.Collections.**
•Puede crear sus propias interfaces, clases, métodos, eventos y delegados genéricos.•Puede limitar las clases genéricas para habilitar el acceso a métodos en tipos de datos determinados.
• Puede obtener información sobre los tipos que se usan en un tipo de datos genérico en tiempo de ejecución mediante la reflexión.

13.2. Métodos genéricos

Los genéricos permiten la reutilización de código en diferentes tipos.
Por ejemplo, declaremos un método que intercambia los valores de sus dos parámetros.**Ejemplo 13.1**

```
static void Swap(ref int a, ref int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

Nuestro método SWAP solo funcionará para parámetros enteros. Si queremos usarlo para otros tipos, por ejemplo, dobles o cadenas, tenemos que sobrecargarlo para todos los tipos con los que queremos usarlo. Además de una gran cantidad de repetición de código, se vuelve más difícil administrar el código porque los cambios en un método significan cambios en todos los métodos sobrecargados.
Los genéricos proporcionan un mecanismo flexible para definir un tipo genérico.

Ejemplo 13.2

```
static void Swap<T>(ref T a, ref T b) {
  T temp = a;
  a = b;
  b = temp;
}
```

En el código anterior, **T** es el nombre de nuestro tipo genérico. Podemos ponerle el nombre que queramos, pero T es un nombre comúnmente usado. Nuestro método SWAP ahora toma dos parámetros de tipo T. También usamos el tipo T para nuestra variable TEMP que se usa para intercambiar los valores.
Tenga en cuenta los pico-paréntesis en la sintaxis $< T >$, que se utilizan para definir un tipo genérico.
Ahora, podemos usar nuestro método SWAP con diferentes.

Ejemplo 13.3

```
static void Swap<T>(ref T a, ref T b) {
  T temp = a;
  a = b;
  b = temp;
}
static void Main(string[] args) {
  int a = 4, b = 9;
  Swap<int>(ref a, ref b);
  //Ahora b es 4, a es 9

  string x = "Hola";
```

```
12    string y = "Mundo";
13    Swap<string>(ref x, ref y);
14    //Ahora x es "Mundo", y es "Hola"
15  }
```

Cuando llamemos a un método genérico, debemos especificar el tipo con el que trabajará usando pico-paréntesis. Entonces, cuando se llama a Swap $< int >$, el tipo T se reemplaza por int. Para Swap $< cadena >$, T se reemplaza por cadena.
Si omite especificar el tipo al llamar a un método genérico, el compilador usará el tipo basado en los argumentos pasados al método.
Se pueden usar múltiples parámetros genéricos con un único método.
Por ejemplo: **Func** $< T, U >$ toma dos tipos genéricos diferentes.

13.3. Clases genéricas

Los tipos genéricos también se pueden usar con las clases.
El uso más común para las clases genéricas es con colecciones de elementos, donde las operaciones como agregar y eliminar elementos de la colección se realizan básicamente de la misma manera, independientemente del tipo de datos almacenados. Un tipo de colección se llama pila. Los elementos se .empujan.º se agregan a la colección y se "quitan.º se quitan de la colección. Una pila a veces se denomina estructura de datos Último en entrar primero (LIFO) por sus siglas en inglés (Last Input First Output).
Ejemplo 13.4

```
1  class Stack<T> {
2    int index = 0;
3    T[] innerArray = new T[100];
4    public void Push(T item) {
5      innerArray[index++] = item;
6    }
7    public T Pop() {
8      return innerArray[--index];
9    }
10   public T Get(int k) { return innerArray[k]; }
11 }
```

La clase genérica almacena elementos en una matriz. Como puede ver, el tipo genérico T se utiliza como el tipo de matriz, el tipo de parámetro para el método **Push** y el tipo de devolución para los métodos **Pop** y **Get**.

Ahora podemos crear objetos de nuestra clase genérica.

Ejemplo 13.5

```
Stack<int> intStack = new Stack<int>();
Stack<string> strStack = new Stack<string>();
Stack<Person> PersonStack = new Stack<Person>();
```

También podemos usar la clase genérica con tipos personalizados, como el tipo PERSON.

En una clase genérica, no es necesario definir el tipo genérico para sus métodos, porque el tipo genérico ya está definido en el nivel de clase.

Los métodos de clase genéricos se llaman igual que para cualquier otro objeto.

Ejemplo 13.6

```
Stack<int> intStack = new Stack<int>();
intStack.Push(3);
intStack.Push(6);
intStack.Push(7);

Console.WriteLine(intStack.Get(1));
//Salida: 6
```

Capítulo 14

Colecciones

14.1. Listas y Diccionarios

.NET Framework proporciona varias clases de colecciones genéricas, útiles para almacenar y manipular datos.
Estas clases están contenidas en el espacio de nombres **System.Collections.Generic**.
La lista es una de las clases de colección comúnmente utilizadas.
Ejemplo 14.1

```
1  List<string> colors = new List<string>();
2  colors.Add("Red");
3  colors.Add("Green");
4  colors.Add("Pink");
5  colors.Add("Blue");
6
7  foreach (var color in colors) {
8    Console.WriteLine(color);
9  }
10 /*Salida: Red Green Pink Blue
11 */
```

Definimos una Lista que almacena cadenas y se itera a través de ella utilizando un bucle **foreach**. La clase **List** contiene una cantidad de métodos útiles.
Add: agrega un elemento a la lista.
Clear: elimina todos los elementos de la lista.
Contains: determina si el elemento especificado está contenido en la Lista.
Count: devuelve la cantidad de elementos en la Lista.
Insert: agrega un elemento en el índice especificado.
Reverse invierte el orden de los elementos en la Lista.

Entonces, ¿por qué usar Listas en lugar de matrices?
Porque, a diferencia de las matrices, el grupo de objetos con el que trabajas en una colección puede crecer y reducirse de forma dinámica.

Los tipos de colecciones genéricas comúnmente usadas incluyen:
Dictionary $< TKey, TValue >$ representa una colección de pares clave / valor que se organizan en función de la clave.
List $< T >$ representa una lista de objetos a los que se puede acceder por índice. Proporciona métodos para buscar, ordenar y modificar listas.
Queue $< T >$ representa una colección de objetos primero en entrar, primero en salir (FIFO).
Stack $< T >$ representa una colección de objetos de último en entrar, primero en salir (LIFO).

Capítulo 15

Concurrencia

15.1. Clases Thread y Parallel

15.2. Tasks

15.3. Sincronización

Capítulo 16

Lambdas, Delegados y Eventos

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

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