Programación C# Yellow Book

Rob Miles

Edición 8.2 Noviembre 2016
Traducción española

C# Yellow Book 2016, edición 8.2 en español

Traducción española por Jesús Ruiz García

DEDICATORIA

A mis padres, Manuel Ruiz Díaz y María García Medina.
Con todo mi cariño y afecto. Os quiero mucho.

A Rob Miles, por autorizarme a traducir su obra. Gracias por todo.

Introducción

Bienvenido

Bienvenido al Maravilloso Mundo de Rob Miles™. En este mundo habitan los chistes malos, los juegos de palabras y por supuesto la programación. En este libro voy a darle algunas nociones sobre el lenguaje de programación C#. Si ya ha programado antes, le estaría muy agradecido si continúa leyendo este libro. Valdrá la pena sólo por los chistes, y por supuesto también porque quizás pueda aprender algo que todavía no sepa.

Si no ha programado antes, no se preocupe. La programación no es una ciencia exacta, es… bueno, programación. Las malas noticias de aprender a programar es que será arrollado por un montón de ideas y conceptos desde el mismo momento en el que se inicia, y esto puede ser confuso. Las claves para aprender a programar son:

Práctica - Realice muchos ejercicios de programación y oblíguese a pensar en las cosas desde la perspectiva de ser un 'solucionador de problemas'.

Estudio - Examine programas realizados por otras personas. Puede aprender mucho estudiando el código que otra gente ha desarrollado. Investigar cómo otra persona realizó el trabajo, es un gran punto de partida para aprender a solucionar problemas. Y recuerde que en muchos casos no hay una *mejor* solución, tan sólo algunas soluciones que resultan ser más adecuadas para un determinado contexto en particular, es decir, la más rápida, la más pequeña, las más fácil de usar, etc.

Persistencia - Escribir programas es un trabajo arduo. Hay que trabajar duro en ello. La principal razón por la que la mayoría de personas no consiguen ser programadores es porque se dan por vencidos en poco tiempo. Esto no ocurre porque no tengan la inteligencia necesaria, sino porque son poco persistentes. Si no ha resuelto un problema de programación en 30 minutos, debería pedir tiempo muerto y buscar ayuda. O al menos alejarse del problema y volver posteriormente a éste. Mantenerse toda la noche tratando de resolver un problema no es una buena idea. Lo único que conseguirá será encontrarse de mal humor a la mañana siguiente. Analizaremos qué hacer cuando todo falle en la sección 5.9.

Leyendo las notas

Estas notas han sido escritas para ser leídas y consultadas cuando lo necesite. Estas contienen una serie de *Puntos de Programación*, basados en la experiencia real de la vida como programador, y debe tenerlas en consideración. También hay algunas partes del texto escritas en una FUENTE ELEGANTE. Estas son muy importantes, y deberá aprendérselas de memoria, como si de música se tratara.

Si tiene algún comentario sobre cómo las notas pueden mejorarse (aunque, por supuesto, lo considero altamente improbable), no dude en contactarme.

Sobre todo, disfrute programando.

Rob Miles

Obtenga una copia de las notas

Estas notas están a libre disposición de los estudiantes de ciencias informáticas de la Universidad de Hull.

El sitio web del libro se encuentra en la siguiente dirección http://www.csharpcourse.com donde también puede encontrar el paquete de diapositivas PowerPoint y los ejercicios de laboratorio del curso de C# en los que se basa este libro. También puede encontrar en este sitio, el código fuente en C# de los ejemplos utilizados en este texto en su versión original en inglés.

Puede obtener una versión electrónica de este libro en su versión original en inglés, para dispositivos Kindle desde Amazon.

Descargar los archivos de código fuente en español

Puede descargar los archivos de código fuente en español de los programas de ejemplo utilizados en este libro, desde mi blog oficial:

https://jesusruizgarcia.wordpress.com/

1 Computadoras y Programas

En este capítulo conocerá lo que es una computadora, y entenderá la forma en la que un programa se comunica con ésta para decirle lo que tiene que hacer. Descubrirá los pasos a seguir cuando se empieza a desarrollar un programa, para obtener un "final feliz" entre usted y su cliente. Por último, echaremos un vistazo a la programación en general y al lenguaje C# en particular.

1.1 Computadoras

Antes de centrarnos en la programación, vamos a conocer un poco mejor a las computadoras. Es importante que hagamos esto, ya que así nos adentraremos mejor en el contexto en el que se sitúan todas las cuestiones relacionadas con la propia programación.

1.1.1 Introducción a las Computadoras

Podríamos describir una computadora como una caja eléctrica que emite zumbidos. Esto, aunque técnicamente sea correcto, puede crear cierta confusión, especialmente entre aquellos que posteriormente intenten programar una nevera. Una forma mejor de describir a una computadora sería la siguiente:

Un dispositivo que procesa una información de acuerdo a una serie de instrucciones que se le ha dado.

Esta definición genérica excluye definitivamente a las neveras, aunque no es tampoco demasiado precisa. Sin embargo, para nuestros propósitos actuales, ésta servirá. Al conjunto unitario de instrucciones que permiten a un ordenador realizar funciones y tareas diversas, se le denomina "programa". En ocasiones, al trabajo de utilizar un programa se le llama erróneamente programación. La mayoría de la gente **no** programa, tan sólo utilizan programas realizados por otras personas. Por lo tanto, debemos hacer una distinción entre los usuarios y programadores. Un usuario realiza un trabajo mediante una computadora que ejecuta el programa adecuado, facilitándole su trabajo. Un programador en cambio tiene un deseo masoquista de juguetear con las tripas de una máquina.

Una de las reglas de oro de la programación es que nunca debe escribir un programa si ya hay uno disponible que hace lo mismo, es decir, un profundo deseo en usted de procesar palabras con una computadora, no debería dar como resultado ¡el que programe un procesador de textos.! No obstante, a menudo se tienen que hacer trabajos con las computadoras que no han sido realizados con anterioridad, y esta es la causa principal por la que la gente estará dispuesta a pagarle para que haga este trabajo, así que vamos a aprender a programar tan bien como ya sabemos utilizar una computadora.

Antes de que entremos de lleno en el divertido negocio de la programación, merece la pena que revisemos algunos términos informáticos:

1.1.2 Hardware y Software

Al comprar una computadora, no solamente recibe una caja que emite zumbidos. La caja, para ser útil, también tiene la suficiente inteligencia incorporada como para entender órdenes sencillas con el fin de realizar tareas. En este punto hay que hacer una distinción entre el software y el hardware de un sistema informático. El hardware es la parte física del sistema. Básicamente si puede golpearlo, y éste deja de funcionar al sumergirlo en un cubo de agua, es hardware. Hardware es la impresionante pila de luces e interruptores que le vendió el vendedor de la esquina.

El software es lo que hace que la máquina funcione. Si una computadora tiene alma, ésta se encuentra y se mantiene en su software. El software emplea la capacidad física del hardware, el cual puede ejecutar programas, para hacer algo de utilidad. Se llama software porque éste no tiene existencia física y es relativamente fácil de modificar. Software es la voz de la computadora de la saga Star Trek.

Todas las computadoras se venden con algún tipo de software incluido. Sin estos, no serían más que un sistema de calefacción novedoso y muy caro. Al software que normalmente viene instalado al comprar una computadora se le conoce como Sistema Operativo. El Sistema Operativo hace que la máquina sea utilizable. Éste se ocupa de toda la información almacenada en el ordenador y proporciona una gran cantidad de comandos que nos permite gestionarla. También le permite ejecutar programas, hayan sido creados por usted o por otras personas. Para desarrollar programas en C# y que funcionen correctamente, tendrá que aprender a comunicarse con el sistema operativo.

1.1.3 Datos e Información

Las personas utilizamos las palabras para intercambiar datos e información. Las máquinas piensan de otra manera y consideran los datos y la información como dos cosas diferentes:

Datos es la colección de on’s y off’s que las computadoras almacenan y manipulan.

Información es la interpretación de los datos que para las personas tienen algún significado. Estrictamente hablando las computadoras procesan datos, los humanos trabajamos con la información. Un ejemplo, la computadora puede tener el siguiente patrón de bits guardado en alguna zona de la memoria:

11111111 11111111 11111111 00000000

Podríamos considerar que este conjunto de bits, significa:

"tiene 256 euros al descubierto en su cuenta bancaria"

o

"se encuentra a 256 pies bajo la superficie de la Tierra"

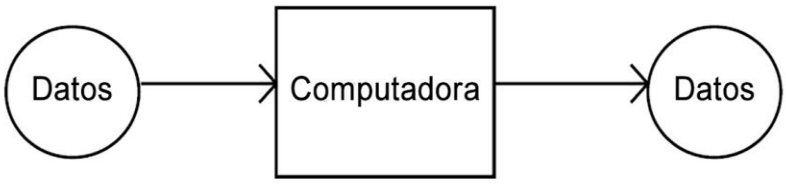
o

"ocho de los treinta y dos interruptores están apagados"

La conversión de los datos en información, es realizada por lo general cuando el ser humano lee la salida de los datos una vez han sido procesados. Se preguntará, ¿Por qué estoy siendo tan insistente con este tema? Lo hago porque es de vital importancia que recuerde que una computadora "no sabe" realmente lo que significan los datos que está procesando. Por lo que a la computadora respecta, los datos son solamente patrones de bits, y es el usuario el que debe darle el significado a esos patrones. ¡Recuerde esto cuando reciba un extracto bancario en donde se le informe que tiene 8.388.608 € en su cuenta!

Procesamiento de datos

Las computadoras son procesadoras de datos. La información es introducida en la computadora; éstas hacen algo con dicha información, y posteriormente generan más información. Un programa informático es el encargado de decirle a la computadora qué hacer con la información recibida. Una computadora trabaja sobre los datos de la misma manera que una máquina que hace salchichas trabaja sobre la carne: se introduce alguna cosa por un extremo, se realiza algún tipo de procesamiento (proceso), y la cosa termina saliendo por el otro extremo:



Un programa no es consciente de los datos que está procesando, de la misma forma que una máquina que hace salchichas no es tampoco consciente de lo que es la carne. Si introduce una bicicleta dentro de la máquina de hacer salchichas, está tratará de obtener salchichas de ella. Introduzca datos sin valor en una computadora, y conseguirá que ésta haga cosas iguales de inútiles. Por tanto, solamente nosotros las personas, somos quienes realmente damos sentido a los

datos (véase más arriba). Para una computadora, los datos son solamente cosas que recibe y que tiene que manipular de alguna manera.

Un programa es tan sólo una secuencia de instrucciones que le dicen a una computadora lo que tiene que hacer con los datos recibidos, y que forma tendrán los datos a la salida una vez hayan sido procesados.

Tenga en cuenta que el procesamiento de datos realizado por las computadoras, es mucho más complejo que leer y escribir números. Estos son algunos ejemplos de los datos que procesan algunas aplicaciones:

Reloj digital: Una micro-computadora en su reloj está tomando y recibiendo los impulsos necesarios que permiten la medición del tiempo y las peticiones realizadas desde los botones, procesando estos datos y mostrándole en pantalla la hora actual.

Coche: Una micro-computadora en el motor de su coche, recibe información de los sensores indicándole la velocidad actual del motor, la velocidad en carretera, la cantidad de oxígeno en el aire, los ajustes del acelerador etc., y produciendo los voltajes de salida que controlan los ajustes del carburador, el avance del encendido etc., lo que ayuda a optimizar el rendimiento del motor.

Reproductor de CD: Una computadora toma una señal del disco y la convierte en el sonido que quiere escuchar. Al mismo tiempo ésta mantiene la cabeza del láser precisamente posicionada y monitorea todos los botones del dispositivo por si quiere seleccionar otra parte del disco.

Videojuegos: Una computadora recibe instrucciones de los controladores, que está utilizando el video jugador para moverse por el mundo artificial que ella misma ha creado.

Tenga en cuenta que algunas de estas aplicaciones de procesamiento de datos no son más que tecnologías aplicadas a los dispositivos existentes para mejorar la forma en la que trabajan. Sin embargo, el Reproductor de CD y los videojuegos, no podrían funcionar si no tuvieran incorporada la capacidad de procesar datos.

La mayoría de los dispositivos razonablemente complejos contienen procesadores de datos que optimizan su rendimiento, y algunos existen solamente gracias a que nosotros podemos crear inteligencia. Es a este mundo al que nosotros, como desarrolladores de software nos estamos dirigiendo. Es importante pensar en el negocio del procesamiento de datos como un trabajo más avanzado que el de calcular las nóminas de los empleados de una empresa: basada prácticamente en leer números, realizar una operación matemática e imprimir los resultados.

Como ingenieros del software es inevitable que pasemos gran parte de nuestro tiempo programando para manejar los componentes destinados a procesar los datos de los distintos dispositivos que tenemos a nuestro alcance. Usted será el encargado de comunicarle a la computadora el trabajo que debe realizar al pulsar un botón. Estos sistemas embebidos hacen que todas las personas utilicen una máquina, ¡sin que realmente ellos sean conscientes de que hay una computadora allí dentro!

También debe recordar que un programa aparentemente inocuo podría tener posibilidades de atentar contra la vida de las personas. Por ejemplo, un médico puede utilizar una hoja de cálculo para calcular las dosis de los fármacos que deben tomar sus pacientes. En este caso, un fallo en el código del programa podría provocar una enfermedad o el fallecimiento de alguno de ellos (no creo que los médicos hagan esto – pero nunca se sabe...)

Punto del Programador: En la base hay siempre hardware

Es importante que recuerde que sus programas son realmente ejecutados por una pieza de hardware que tiene limitaciones físicas. Debe asegurarse, que el código que escriba realmente se ajusta a las capacidades de la máquina a la que está destinada y que opera a una velocidad razonable. La potencia y la capacidad de las computadoras modernas hacen de éste, un problema menos importante que en el pasado, pero debe ser consciente de estos aspectos. Hablaré sobre estos detalles cuando lo considere oportuno.

1.2 Programas y Programación

La programación es un arte oscuro. Esta es de la clase de cosas que a regañadientes uno admite hacer por la noche con las persianas bajadas y sin que nadie le vea. Dígale a la gente que es programador y obtendrá una de las siguientes reacciones o respuestas:

1. Una mirada fija perdida como señal de ausencia.
2. “Eso suena interesante…”, seguido de una larga explicación sobre la ventana de doble acristalamiento que ellos acaban de instalar.
3. Será interrogado por cómo resolver todos los problemas informáticos que ellos han tenido a lo largo de toda su vida.
4. Una mirada que indica que tú no puedes ser muy bueno en tu trabajo, ya que todos los programadores conducen Ferraris y acceden ilegalmente al Banco de Inglaterra cada vez que quieren.

La programación es definida por la mayoría de la gente como una manera de ganar grandes sumas de dinero haciendo algo que nadie puede entender.

La programación es definida por mi como derivar y expresar una solución a un problema dado, de forma que un sistema informático pueda entenderla y ejecutarla.

Una o dos cosas se deducen de esta última definición:

- Tiene que resolver el problema antes de poder programarlo.
- La computadora tiene que entender los que usted está indicándole que haga.

1.2.1 ¿Qué es un Programador?

¡A mí me gusta pensar que un programador es como un fontanero! Un fontanero llega al trabajo con una gran bolsa de herramientas llena de piezas de repuesto. Tras mirar por un tiempo, chasquea la lengua en señal de desaprobación, abre su bolsa y saca varias herramientas y piezas, ajustándolas de manera precisa para solventar el problema. La programación es básicamente esto. Usted tiene un problema que resolver y tiene a su disposición una gran bolsa de trucos de programación, en este caso determinados por un lenguaje de programación. Usted estudiará y considerará el problema durante un tiempo hasta encontrar la manera de resolverlo, y a continuación, adaptará la solución al lenguaje elegido para solventar el problema que tiene. El secreto de la programación se basa en conocer que partes necesita utilizar de su bolsa de trucos para resolver cada parte del problema.

Del Problema al Programa

El arte de tomar un problema y dividirlo en una serie de instrucciones que pueda darle a una computadora es la parte más interesante de la programación. Desafortunadamente, también es la parte más dura de la misma. Si piensa que aprender a programar es simplemente una cuestión de aprender un lenguaje de programación está muy equivocado. De igual forma, si piensa que la programación es simplemente una cuestión de buscar un programa que resuelva un problema ¡está doblemente equivocado!

Hay muchas cosas que debe considerar cuando escribe un programa; y no todas ellas están directamente relacionadas al problema en cuestión en el que está trabajando. Empezaré suponiendo que tiene que desarrollar un programa para un cliente. Él o ella tiene un problema y le gustaría que usted le creara un programa para resolverlo. ¡Asumiremos que el cliente no sabe nada sobre computadoras!

Inicialmente no debemos hablarle sobre el lenguaje de programación que vamos a utilizar, los tipos de computadoras existentes o temas similares a estos; simplemente debemos asegurarnos de entender lo que el cliente quiere que hagamos.

Resolviendo el problema equivocadamente

Aparecer con una solución perfecta a un problema que el cliente no tiene, es algo que sucede con sorprendente frecuencia en el mundo real. Muchos proyectos de software han fracasado porque el problema que resolvían era el equivocado. Los desarrolladores del sistema simplemente no se enteraron de lo que realmente se requería, y en su lugar crearon lo que ellos pensaban que se les había solicitado. Los clientes asumieron que como los desarrolladores habían dejado de hacerles preguntas, la verdadera solución al problema requerido estaba siendo desarrollada, y solamente a

la entrega final descubrieron la horrible realidad. Por eso es muy importante que un programador no haga nada hasta que sepa exactamente lo que se requiere.

Esto es una especie de autodisciplina. Los programadores se enorgullecen de su capacidad para encontrar soluciones, por lo que tan pronto como se les da un problema comienzan de inmediato a pensar en distintas maneras de resolverlo, esto es casi un acto reflejo. Lo que debe de hacer es pensar "¿He entendido realmente cuál es el problema que tengo que resolver?". Antes de ponerse a programar, debe asegurarse de tener una definición irrefutable de cuál es el problema, en la que tanto usted como su cliente estén de acuerdo y conformes.

En el mundo real tal definición es denominada en ocasiones como Especificación de Diseño Funcional o FDS. Esta especificación define exactamente lo que realmente quiere el cliente. Tanto usted como el cliente deben firmarla, indicando en la línea final que si proporciona un sistema que se comporta de acuerdo con las especificaciones de diseño, el cliente deberá pagarle. Una vez que usted tiene su especificación de diseño, entonces puede pensar en las diferentes maneras que tiene para resolver el problema. Podría pensar que este proceso no es necesario si está desarrollando un programa para sí mismo; ya que no existe un cliente al que tengamos que satisfacer. **Esto no es cierto**. Escribir algún tipo de especificación le obliga a pensar en su problema a un nivel muy detallado. Además, también le obliga a pensar sobre lo que su sistema no debe de hacer, y a establecer las expectativas que tiene el cliente desde el principio.

Punto del Programador: La especificación debe estar siempre ahí

A lo largo de mi vida, he desarrollado muchos programas. **Nunca** he creado un programa sin tener antes de nada una especificación sólida sobre lo que requiere el cliente. Esto es cierto incluso (o quizás especialmente), si voy a realizar este trabajo para un amigo.

Las técnicas modernas de desarrollo ponen al cliente directamente en el corazón del desarrollo, y los involucra a ellos en el proceso de diseño. Este trabajo que sobre la base es muy duro (y en realidad no es tan útil) es esencial para obtener una especificación definitiva al inicio de un proyecto. Usted como desarrollador no sabrá seguramente demasiado sobre el negocio de su cliente, y ellos como clientes no conocerán las limitaciones y posibilidades de la tecnología. Teniendo esto en cuenta, es una buena idea hacer una serie de versiones previas de la solución y discutir cada una de ellas con el cliente antes de pasar a la siguiente. Este proceso es denominado *prototipo* o *modelo de prototipos*.

1.2.2 Un problema sencillo

Considere la siguiente situación; usted se encuentra sentado en su sillón favorito del pub al que suele asistir habitualmente contemplando el universo, cuando es interrumpido en su ensimismamiento por un amigo suyo que se dedica al negocio de la fabricación y venta de ventanas

de doble acristalamiento para ganarse la vida. Él sabe que usted es un buen programador y le gustaría contar con su ayuda para resolver un problema que tiene: Él acaba de empezar a ofrecer sus propias medidas "personalizadas" de ventana, y está buscando un programa que realice el cálculo del costo de los materiales. Él quiere simplemente introducir las dimensiones de la ventana y tras realizar dicha acción, obtener una copia impresa de los costos de fabricación de la ventana, en cuanto a la cantidad de madera y vidrio requerido. Usted piensa "Este parece un buen trabajo para ganar algo de dinero", y una vez acordado un precio empezará a trabajar en el asunto. La primera cosa que necesita hacer es averiguar exactamente lo que el cliente quiere que haga…

Especificando el Problema

Cuando se considera cómo escribir la especificación de un sistema hay tres cosas importantes a tener en cuenta:

- La información que fluye a través del sistema
- La información que fluye a la salida del sistema
- Lo que el sistema hace con la información.

Hay un montón de maneras de representar esta información en forma de diagramas, a continuación, escribiremos el texto de cada una de las etapas que forman el diseño de una especificación.

Información de entrada

En el caso de nuestro imperecedero problema de la ventana de doble acristalamiento, podemos describir esta información como:

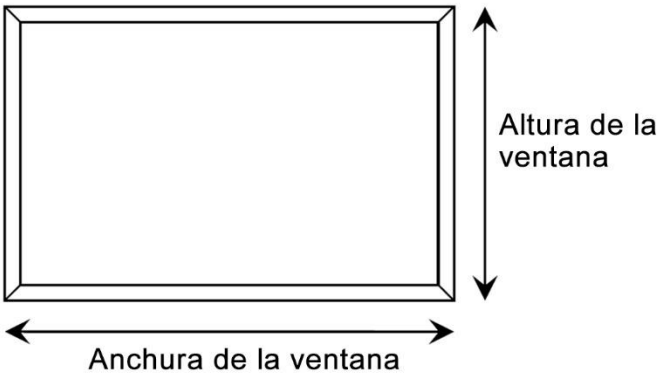
- La anchura de la ventana
- La altura de la ventana.

Información de salida

La información que nuestro cliente quiere conocer es:

- el área de vidrio requerida para la ventana
- la longitud de la madera requerida para construir un marco.

Puede visualizar lo que necesitamos si echa un vistazo al siguiente diagrama:



El área del cristal es el ancho multiplicado por la altura. Para hacer el marco se necesitan dos piezas de madera correspondientes a la anchura de la ventana, y otros dos tableros de madera correspondientes a la altura de la ventana.

Punto del Programador: Los metadatos son importantes

Los datos que describen a otros datos son llamados ***metadatos***. La palabra meta en esta situación implica un "paso atrás" en el problema, para que puedan ser considerados en un contexto más amplio. En el caso de nuestro programa de la ventana, los metadatos nos darán mayor información sobre los valores que están siendo utilizados y producidos, específicamente las unidades en las que la información es expresada y el rango válido de valores que los datos deben tener. Para cualquier valor que usted represente en un programa, debe tener al menos este nivel de metadatos.

Lo que realmente hace el programa

El programa puede derivar los dos valores necesarios de acuerdo con las siguientes ecuaciones:

```
area vidrio = anchura ventana * altura ventana
longitud madera = (anchura ventana + altura ventana) * 2
```

Entrando en mayor detalle

Ahora tenemos una comprensión bastante buena de lo que el programa va a realizar por nosotros. Siendo sensatos y en contra de lo que la gente piensa, nosotros no nos detenemos en este punto. Ahora tenemos que preocuparnos sobre cómo nuestro programa decidirá si la información de entrada que recibe el programa es realmente válida o no lo es.

Esto debe hacerse conjuntamente con el cliente, él o ella debe entender que, si la información es dada dentro de los rangos especificados, su programa considerará los datos como válidos y actuará en consecuencia.

En el caso anterior, podríamos por tanto ampliar la definición de los datos de entrada como sigue:

- La anchura de la ventana especificada en metros, y siendo válidos los valores comprendidos entre 0.5 metros y 3.5 metros inclusive.
- La altura de la ventana especificada en metros, y siendo válidos los valores comprendidos entre 0.5 metros y 2.0 metros inclusive.

Tenga en cuenta que también hemos añadido unidades a nuestra descripción, esto es muy importante - quizá nuestro cliente compra la madera a un distribuidor que las venda por pies, en cuyo caso nuestra descripción de salida será la siguiente:

- El área de vidrio requerido para la ventana, en metros cuadrados. Recuerde que nosotros estamos vendiendo doble acristalamiento, por lo que se requerirán dos paneles.
- La longitud de madera necesaria para el marco, dada en pies usando el factor de conversión de 3,25 pies por metro.

Después de haber escrito todo esto de una forma en la que tanto usted como el cliente puedan comprenderla, ambas partes debéis firmar la especificación final realizada, y es entonces cuando puede comenzar a realizar el trabajo.

Comprobando que el programa funciona

En un mundo real ahora haría una prueba que le permita comprobar y verificar que el programa funciona, por ejemplo, podría decir:

“Si yo introduzco en los datos de entrada del programa 2 metros de altura y 1 metro de ancho, el programa debería informarme que yo necesito 4 metros cuadrados de vidrio y 19.5 pies de madera.”

Un método de ensayo que ha sido diseñado adecuadamente para un proyecto, debe poner a prueba todos los posibles estados por los que puede pasar un programa, incluyendo las condiciones de error que son de suma importancia. En un sistema, la persona que escribe el programa tendrá que crear algunas pruebas de software automatizadas; lo que consiste en utilizar un software especial (casi siempre separado del software que se prueba) para controlar la ejecución de pruebas y así poder comparar entre los resultados obtenidos y los resultados esperados. Tanto el cliente como el proveedor deben estar de acuerdo en el número y tipos de pruebas a realizar, debiendo firmar un documento que las describa.

Las pruebas de software (testing) son muy importantes cuando estamos desarrollando una aplicación. Hay incluso una técnica de desarrollo, en el que se escribe las pruebas *antes* de escribir el programa que realmente hace el trabajo. Esta técnica es conocida como Desarrollo guiado por pruebas de software, o Test-driven development (TDD). Utilizar esta técnica es realmente una buena idea, así que la veremos un poco más adelante. En términos de producción de código, usted debe esperar a escribir tanto o más código para testear su solución, como para la propia solución en sí misma. Recuerde esto a la hora de calcular cuánto trabajo supone el realizar una tarea concreta.

Recibiendo el pago

Llegado a este punto el proveedor sabe que, si el sistema ha pasado todos todas las pruebas, ¡el cliente no tendrá más remedio que pagar por el trabajo que hemos realizado! Tenga en cuenta también que, aunque el diseño y los métodos de ensayo hayan sido congelados, no hay ninguna ambigüedad que puede llevar a que el cliente solicite cambios en nuestro trabajo, ¡aunque por supuesto esto todavía puede suceder!

Las buenas noticias para el desarrollador son que, si se le solicita algún tipo de modificación en el software, estos cambios deben ser considerados como un trabajo adicional, por los que deberán ser pagados a parte por su cliente.

Implicando al cliente

Tenga en cuenta también que en un sistema "apropiado", el cliente esperará a que se le consulte sobre cómo el programa va a interactuar con el usuario, ¡a veces incluso hasta sobre el color de las letras que aparecerán en la pantalla! Recuerde que una de las cosas más peligrosas que un programador puede pensar es "¡Esto es lo que él quiere!". La precisa interacción con el usuario, lo que el programa hace cuando se produce un error, cómo se presenta la información en pantalla etc., son cosas sobre las que le garantizo que el cliente va a tener una opinión firme. Lo ideal sería que toda esta información sea escrita y detallada en la especificación, incluyendo los diseños de las pantallas y detalles de las teclas a presionar en cada etapa. Muy a menudo los prototipos se utilizarán para tener una idea de cómo el programa debe verse y cómo debería sentirse el cliente al utilizarlo.

Si con esto le parece que le está pidiendo al cliente a que le ayude a desarrollar el programa, ¡entonces está en lo correcto! Su cliente espera que usted haya tomado la descripción del problema y vaya a su habitáculo particular - para regresar posteriormente con la solución perfecta. Pero esto no va a suceder. Lo que ocurrirá es que usted volverá con un trabajo aproximado al 60% de lo idealizado por el cliente. El cliente le dirá con qué partes del programa está conforme y cuáles de ellas quiere que modifique. Usted volverá a su habitáculo particular, murmurando en voz baja entre dientes, y regresará con otro sistema que deberá ser aprobado. De nuevo, la ley de Rob dice que

de un 60% de las deficiencias modificadas, un 40% habrán sido corregidas con éxito, así pues, deberá aceptar de nuevo los cambios propuestos por su cliente y regresar de nuevo a su teclado….

El cliente cree que esto es genial, y esta situación le evocará el recuerdo de un sastre de lujo que consigue confeccionar el perfecto ajuste a medida tras realizar numerosas modificaciones a la prenda. Todo lo que el cliente hace es considerar alguna cosa, sugerir cambios y esperar a que usted regrese con la siguiente versión del programa para encontrar si hay alguna otra cosa que no le termine de convencer. Ellos podrían sentirse un poco molestos si llegando el plazo de entrega todavía no ha aparecido con el producto final terminado, aunque ellos siempre podrían volver a recuperar sus estados de ánimo demandándole.

Ahora, hemos cerrado el círculo, porque como mencioné anteriormente la elaboración de un prototipo es una buena manera de construir un sistema cuando no tiene clara la especificación inicial. No obstante, debe tener en cuenta que, si va a utilizar prototipos, es mejor que los planifique y estudie bien desde el principio, para no tener que terminar haciendo un trabajo extra a causa de que su comprensión inicial del problema era errónea.

Si con su fuerte insistencia sobre la elaboración de la especificación, logra que el cliente piense exactamente sobre lo que se supone debe hacer el programa y cómo va a funcionar, todavía mejor. El cliente podría decirle “Pero yo te estoy pagando a ti porque eres el experto en computadoras, Yo no sé nada sobre estas máquinas”. Esto no es una excusa. Explíquele los beneficios de la filosofía “Hacer las cosas bien desde el principio”, y si esto no funciona ¡saque un revolver y oblíguele a aceptar la especificación!

Una vez más, si yo tuviera que subrayar algo en rojo para destacarlo sería que: Todo lo anteriormente explicado se aplica también incluso si usted está desarrollando el programa para sí mismo. ¡Usted es su peor cliente!

Debe pensar que estoy insistiendo demasiado sobre este asunto, cuando probablemente considere que no es tan necesario. Al fin y al cabo, pensará que cuando empezamos a programar vamos a realizar programas más bien simples y poco triviales, por lo que le parecerá que las técnicas descritas anteriormente son demasiado pesadas y nos llevarán mucho tiempo realizarlas. Usted está equivocado. Una muy buena razón para realizar este tipo de técnicas, es que obtendrá la mayor parte del programa descrito detalladamente (muchas veces con la ayuda de su cliente), por lo que esto le facilitará la transcripción que realizamos al lenguaje de programación. Esto es lo que sabemos ahora que tiene que hacer nuestro programa de doble acristalamiento:

```
introducir la anchura
verificar el valor
introducir la altura
verificar el valor
```

calcular dos veces el area del vidrio necesaria e imprimir ésta
calcular la longitud de la madera necesaria: (anchura + altura) * 2 * 3.25 e imprimir el resultado

Su parte de trabajo como programador, es ahora convertir la descripción anterior a un lenguaje que pueda ser utilizado en una computadora…

Punto del Programador: Los buenos programadores son buenos comunicadores

El arte de hablar con un cliente y saber lo que él o ella necesita es simplemente eso, un arte. Si quiere ser un buen programador tendrá que aprender a hacer esto. Una de las primeras cosas que debe hacer es acabar con la idea de "Yo estoy escribiendo un programa para un cliente" y sustituirla por "Nosotros estamos creando una solución a un problema". Usted no trabaja para sus clientes, usted trabaja con ellos. Es muy importante tener esto en cuenta, particularmente cuando tenga que tratar con el cliente sobre cuestiones de funcionalidades o precios.

1.3 Lenguajes de programación

Una vez que sabemos lo que el programa debe hacer (especificación), y cómo vamos a determinar si este funciona correctamente o no (pruebas o testeo); ahora tenemos que expresar nuestro programa en un formato en el que la computadora pueda trabajar con esta información. Probablemente se preguntará "¿Por qué necesitamos los lenguajes de programación?, ¿por qué no podemos usar el inglés?":

1. Las computadoras son demasiado tontas, así que no pueden entender el inglés.
2. El inglés sería un lenguaje de programación pésimo.

Con respecto al primer punto, por ahora no se pueden fabricar computadoras más inteligentes. Las computadoras son inteligentes gracias al software que incluimos dentro de ellas, y hay limitaciones en cuanto al tamaño de los programas que podemos crear y la velocidad con las que pueden comunicarse con nosotros. Aunque por ahora no podemos hacer que una computadora entienda el inglés, sí podemos conseguir que sea capaz de entender un lenguaje muy limitado que nosotros utilizaremos para ordenarle a ésta lo que debe hacer.

Con respecto al segundo punto. El inglés como idioma que es, está repleto de ambigüedades. Es muy difícil expresar algo de manera inequívoca utilizando el inglés. Si no me cree, ¡pregúntele a un abogado!

Los lenguajes de programación consiguen resolver ambos problemas. Ellos son lo suficientemente simples como para ser entendidos por una computadora, y además reducen la ambigüedad de nuestro idioma.

Punto del Programador: El lenguaje no es lo importante

Existen una gran cantidad de lenguajes de programación, y durante su carrera usted aprenderá más de uno. C# es un gran lenguaje para iniciarse en la programación, pero no piense que éste será el único lenguaje que tendrá que aprender.

1.4 C#

En este libro vamos a aprender un lenguaje de programación llamado C# (pronunciado como C sharp). ¡No cometa el error de llamar al lenguaje C almohadilla, ya que demostrará su ignorancia de inmediato! C# es un lenguaje de programación flexible y potente con una historia detrás interesante. Éste fue desarrollado por Microsoft Corporation por varias razones, algunas técnicas, otras políticas y otras varias relacionadas con el marketing.

C# tiene grandes similitudes con los lenguajes de programación C++ y Java, tomando prestadas (o implementando mejoras) de las características de estos lenguajes. Los orígenes tanto de Java como de C++ se remontan al pasado, concretamente a un lenguaje de programación llamado C, un lenguaje potente y altamente peligroso creado en la década de 1970. C es especialmente famoso por ser el idioma en el que fue escrito el sistema operativo UNIX, siendo diseñado especialmente para éste.

1.4.1 Peligrosidad de C

Me referí a C como un lenguaje *peligroso*. ¿Qué quise decir con esto? Pensemos por un momento en una motosierra. Si yo, Rob Miles, tengo la necesidad de utilizar una motosierra, lo primero que tendría que hacer sería adquirirla en una tienda. Como no tengo ningún tipo de experiencia utilizando una motosierra, probablemente esperaría que ésta viniera con un montón de características y medidas de seguridad incorporadas, tales como seguros, protectores y frenos automáticos. Éstas la harán mucho más segura, aunque a causa de estas medidas de seguridad probablemente tendría limitada la herramienta y yo no podría cortar ciertos tipos de árbol. Si yo fuera maderero, podría adquirir una motosierra profesional que no trajese incorporada ninguna característica de seguridad y que por tanto me sirviera para cortar casi cualquier cosa. La cuestión es que si cometo un error con la herramienta profesional sin limitaciones podría fácilmente perder la pierna, algo que la herramienta limitada no permitiría que sucediera.

En términos de programación, lo que esto significa es que C carece de algunas características de seguridad proporcionadas y previstas por otros lenguajes de programación. Esto hace al lenguaje mucho más flexible.

Por tanto, si yo hago algo estúpido, C no me detendrá, así que yo tengo muchas más posibilidades de bloquear, provocar cierres inesperados o cuelgues de la aplicación o el sistema, programando bajo C que si utilizo un lenguaje de programación más seguro.

Punto del Programador: Las computadoras son tontas

Considere que siempre se debe trabajar sobre la base de que una computadora no tolera errores por su parte, y que si realiza algo estúpido ¡puede causar un desastre! Pensando así logrará concentrarse de manera adecuada.

1.4.2 Seguridad de C#

El lenguaje C# incorpora lo mejor de ambos mundos a este respecto. Un programa C# puede contener partes *administradas* o *no administradas*. El código administrado no corre directamente sobre el sistema operativo que lo está ejecutando. Esto le asegura de que sea difícil (aunque probablemente no imposible), bloquear o provocar el cuelgue del sistema ejecutando código administrado. Sin embargo, todas estas ventajas tienen un precio, causando que sus programas sean más lentos.

Para obtener el máximo rendimiento posible, y permitir el acceso directo a partes fundamentales del sistema, puede marcar su programa como no administrado. Un programa no administrado es más rápido, pero si se cuelga es capaz de afectar al sistema de la computadora por lo que ésta podría no responder correctamente ante esta situación. El cambio a modo no administrado es análogo a quitar el seguro para el gatillo del acelerador del protector de mano de su nueva motosierra porque éste se opone en su camino.

C# es un gran lenguaje para comenzar a trabajar con partes administradas, ya que, gracias a esto, le será más fácil entender lo que sucede en caso de que su programa no funcione correctamente.

1.4.3 C# y Objetos

C# es un lenguaje *orientado a objetos*. Los objetos son un mecanismo de organización que le permite dividir su programa en partes sensibles, encargándose cada una de ellas de una parte del sistema. El Diseño Orientado a Objetos hace que los grandes proyectos sean mucho más fáciles de

diseñar, probar (testear) y extender/escalar (ampliar). Éste también le permite crear programas que pueden tener un alto grado de fiabilidad y estabilidad.

Estoy muy interesado en la programación orientada a objetos, pero no le diré nada sobre ésta de momento. Y no lo hago porque no sepa mucho sobre este tema (honestamente), sino porque creo que hay algunos aspectos fundamentales relacionados con la programación, que deben ser abordados antes de que podamos utilizar objetos en nuestros programas.

El uso de objetos es una cuestión tanto de diseño como de programación, por tanto, tenemos que saber programar antes de que podamos diseñar sistemas más grandes y complejos.

1.4.4 Ejecutar programas escritos en C#

C# es un lenguaje de programación *compilado*. La computadora no puede entender el lenguaje directamente, por lo que un programa llamado *compilador* convierte el texto C# en instrucciones de bajo nivel que son mucho más simples. Estas instrucciones de bajo nivel son convertidas a su vez en comandos reales que maneja el hardware que ejecuta su programa.

Veremos con más detalle cómo funcionan los programas C# un poco más adelante, por ahora lo que tiene que recordar es que necesita exponer su maravilloso programa desarrollado en C# ante el compilador, antes de que realmente pueda verlo funcionar.

Un compilador es un programa especial que sabe cómo decidir si su programa es legítimo. La primera cosa que éste realiza es comprobar si hay errores en la forma en que ha utilizado el propio lenguaje. Si el compilador no encuentra errores generará una salida expresada en el lenguaje objeto.

El compilador también podría indicar advertencias, las cuales se producen cuando reconoce que ha hecho algunas cosas que no son técnicamente incorrectas, pero que no le parecen del todo lógicas. Un ejemplo de situación de advertencia es cuando declaramos algún elemento, pero no usamos ese elemento que hemos creado en ninguna parte del programa. El compilador le advierte sobre esto, por si ha olvidado utilizar ese elemento en su programa.

El lenguaje C# se suministra con otro montón funcionalidades que permiten a los programas desarrollados en este lenguaje, hacer cosas como leer entradas de texto desde el teclado, tomar una captura de la pantalla, configurar y establecer conexiones de red y similares. Estas características adicionales están disponibles para que las pueda utilizar también en su programa C#, pero deberá solicitarlas explícitamente. Estas características adicionales, serán localizadas automáticamente cuando se ejecute el programa. Más adelante veremos cómo puede dividir un programa en distintas partes, de manera que pueda haber varios programadores trabajando en el desarrollo de un mismo programa.

1.4.5 Creación de programas en C#

Microsoft ha desarrollado una herramienta llamada Visual Studio, que es un gran entorno de desarrollo integrado (IDE) que incluye un conjunto de herramientas destinadas para desarrollar programas. Este entorno consta de un compilador, un editor integrado, y un depurador. Existen distintas versiones de Visual Studio cada una de ellas con un conjunto de características diferentes. Entre estas versiones, hay una versión gratuita denominada Visual Studio Express Edition (ahora denominado Visual Studio Community), la cual es una buena versión para comenzar. Otro recurso gratuito es el llamado Microsoft .NET Framework. Este marco de trabajo, proporciona una gran cantidad de herramientas de línea de comandos, es decir, comandos que puede escribir para que los ejecute el símbolo del sistema, y que puede utilizar para compilar y ejecutar sus programas en C#. Aunque en este punto, hay que aclarar que es usted el que deberá decidir cómo crear y ejecutar sus programas.

No voy a entrar en detalles de cómo descargar e instalar el .NET framework; eso lo dejo para otros libros de texto o documentos que puede encontrar en la red. Así pues, asumiré que está utilizando una computadora que cuenta con un editor de texto (generalmente el Bloc de notas) y el .NET framework instalado.

La computadora humana

Desde luego inicialmente es mejor plasmar sus programas sobre papel. Soy de la opinión de que se escriben mejores programas cuando uno no está sentado frente a la computadora, es decir, creo que el mejor enfoque consiste en escribir (o al menos trazar) su solución en papel la más alejado posible de la máquina. Una vez que se está sentado delante del teclado, se apodera de nosotros una gran tentación de empezar a presionar teclas y escribir directamente sin analizar algún código que pensamos que podría funcionar. Esto no es buena técnica. Probablemente a base de intentos, conseguiría que funcione, pero perderá mucho tiempo eliminando líneas de código innecesarias y corrigiendo otras líneas de código que hacen que el programa no funcione del todo bien.

Si se sienta y plasma en un papel utilizando un lápiz la primera solución, probablemente obtendrá un sistema funcional en la mitad de tiempo.

Punto del Programador: Los buenos programadores depuran menos

No me impresionan aquellos programadores que pasan días enteros utilizando las terminales de trabajo, estableciendo una lucha sin descanso con programas enormes para conseguir depurarlos de forma adecuada. Me impresionan aquellos que nada más aparecer, teclean el código del programa ¡y hacen que éste funcione a la primera!

1.4.6 ¿Qué comprende un programa C#?

Si su madre quiere explicarle cómo hacer su pastel de frutas favorito, ella escribirá la receta en un trozo de papel. La receta constará de una lista de los ingredientes necesarios, seguido por una secuencia de acciones a realizar con estos productos.

Un programa puede ser considerado como una receta, pero estará escrita para que la siga una computadora, no un cocinero. Los ingredientes serán los valores (denominadas *variables*) con los que quiere que su programa trabaje. El programa en sí será una secuencia de acciones (llamadas *sentencias*) que deben ser seguidas por el ordenador. En lugar de escribir el programa en un trozo de papel, deberá hacerlo en un archivo de la computadora denominado archivo de *código fuente*.

Así es como el compilador actúa. Un archivo de código fuente contiene tres cosas:

- instrucciones para el compilador.
- información sobre las estructuras que almacenarán los datos a almacenar y manipular.
- instrucciones que manipulan los datos.

Realizará estos pasos en orden:

Toma de control por parte del compilador

El compilador de C# necesita conocer ciertos aspectos sobre su programa. Necesita saber qué recursos externos va a utilizar su programa. Además, éste también puede informarle de opciones para el desarrollo de su programa que considera importantes. Algunas partes de su programa le proporcionarán esta información al compilador indicándole lo que debe hacer.

Almacenar los datos

Los programas trabajan procesando datos. Los datos tienen que ser almacenados en la computadora mientras el programa los procesa. Todos los lenguajes de programación soportan *variables* de una u otra forma. Una variable es un espacio en el sistema de almacenaje (memoria principal de una computadora) y un nombre simbólico (un identificador) que está asociado a dicho espacio en el cual un valor se almacena mientras el programa se encuentre ejecutándose. C# también le permite construir *estructuras* que pueden contener más de un elemento, por ejemplo, una sola estructura podría almacenar toda la información relacionada con un cliente en particular de un banco. Como parte del proceso de diseño del programa, tendrá que decidir cuáles elementos de estos datos necesitan ser almacenados. También deberá decidir sobre los nombres simbólicos que usará para identificar estos elementos.

Describir la solución

Las instrucciones reales que describen la solución al problema, deben formar parte de su programa. Una única y simple instrucción que realiza una operación, en un programa C# es denominada *sentencia*. Una sentencia es una instrucción que realiza una operación concreta, por ejemplo, sumar dos números y almacenar el resultado.

Lo realmente apasionante sobre la programación es que algunas sentencias pueden alterar el orden de ejecución de las instrucciones. Es decir, su programa puede tomar decisiones sobre que instrucción será la siguiente en ser ejecutada dependiendo del código. C# puede agrupar diferentes sentencias para formar un fragmento de código que realiza una tarea en particular. A este fragmento de código se le denomina *método*.

Un método puede ser muy pequeño o muy grande. Un método puede devolver un valor que puede o no ser de interés. Su programa puede contener tantos métodos como considere necesario, y por supuesto puede darles a éstos cualquier nombre que desee. Además, un método puede hacer referencia hacía otros métodos. El lenguaje C# también cuenta con un gran número de *bibliotecas* disponibles que puede utilizar. Éstas le salvan de tener que "reinventar la rueda", cada vez que escribe un programa. Analizaremos los métodos de una manera más profunda un poco más adelante.

Identificadores y palabras clave

Debe dar un nombre a cada método que cree, ajustando este nombre a lo que la función realiza, por ejemplo, `MostrarMenu` o `GuardarEnArchivo`. En todo programa C#, existe un método especial llamado `Main` que es desde el que se inicia la ejecución del programa. Los nombres que usted inventa para identificar elementos dentro del código de su programa son denominados *identificadores*. También puede crear identificadores cuando tenga que almacenar valores de estos elementos; una variable denominada `longitudMadera` podría ser una buena elección para almacenar el valor de la longitud de la madera requerida. Más adelante veremos las reglas y convenciones que debe tener en cuenta al crear identificadores.

Las palabras que forman parte del propio lenguaje C# son denominadas *palabras reservadas*. En una receta una palabra reservada podría ser "mezclar" o "calentar" o "hasta". Éstas le permitirían decir cosas tales como "***calentar*** *azúcar* ***hasta*** *fundirla*" o "***mezclar hasta*** *suavizar*". En efecto, se dará cuenta a medida que avance que los programas tienen las mismas similitudes de elaboración que una receta. Las palabras reservadas aparecerán de color azul en el texto de este libro.

Objetos

Algunos de los elementos que escribimos en los programas son objetos, que son parte del framework (marco de trabajo) que nosotros estamos utilizando. Para continuar con nuestra

analogía de la cocina, estos elementos son utensilios como recipientes de mezcla y horneado, que se utilizan durante el proceso de cocción. Los nombres de los objetos serán presentados en un tono diferente de azul en algunos de los listados de este libro.

Mostrar mensajes de texto en un programa

Hay dos tipos de texto que pueden aparecer en un programa. Las instrucciones que utiliza para indicar lo que quiere llevar a cabo, y las cadenas de texto que desea que el programa muestre al usuario. Su madre podría añadir la siguiente instrucción para su receta de la tarta: Ahora dibuja las palabras “Feliz Navidad” sobre la tarta en glaseado rosa.

Ella está utilizando comillas dobles para marcar o señalar el texto que se va a dibujar en la tarta, y C# trabaja exactamente de la misma manera. En un programa, el texto “Feliz Navidad” no forma parte de las instrucciones; ya que este texto ha sido escrito para ser visualizado posteriormente por pantalla. Este tipo de mensajes son presentados en color rojo en este texto.

Colores y Convenciones

Los colores que utilizo en este libro son intencionados para que se correspondan con los colores que verá cuando edite sus programas utilizando un editor de programación profesional, como el suministrado como parte de *Visual Studio*. Los colores sirven para hacer los programas más fáciles de entender, pero no tienen realmente un significado especial. Estos colores son añadidos automáticamente por el editor a medida que escribe su programa.

2 Procesamiento de datos simple

En este capítulo crearemos un programa realmente útil (especialmente si se dedica al negocio del doble acristalamiento). Comenzaremos creando una solución muy sencilla y estudiando las sentencias de C# que realizan el procesamiento básico de los datos. Seguidamente utilizaremos características adicionales del lenguaje C# para mejorar la calidad de la solución.

2.1 Nuestro primer programa en C#

El primer programa que vamos a ver leerá la anchura y la altura de una ventana, e imprimirá la cantidad de madera y vidrio requerida para fabricar una ventana que quepa en un hueco de ese tamaño. Este es el problema que nos propusimos resolver en la sección 1.2.2

2.1.1 El Programa de ejemplo

La mejor manera de empezar a conocer C# es realizando nuestro primer programa:

```
using System;

class CalcularAcristalamiento
{
     static void Main()
     {
          double anchura, altura, longitudMadera, areaVidrio;
          string cadenaAnchura, cadenaAltura;

          cadenaAnchura = Console.ReadLine();
          anchura = double.Parse(cadenaAnchura);

          cadenaAltura = Console.ReadLine();
          altura = double.Parse(cadenaAltura);

          longitudMadera = 2 * ( anchura + altura ) * 3.25;

          areaVidrio = 2 * ( anchura * altura );

         Console.WriteLine ( "La longitud de la madera es" +
                longitudMadera + " pies" );
         Console.WriteLine( "El área del vidrio es " +
                areaVidrio + " metros cuadrados" );
         Console.ReadKey();
     }
```

```
}
```

Código de Ejemplo 01 Calcular Acristalamiento

Este es un programa legítimo. Si le entregara este programa a un compilador de C#, compilaría sin problemas y podría ejecutarlo. El trabajo realmente importante de este programa lo realizan las siguientes dos líneas del programa:

```
longitudMadera = 2 * (anchura + altura) * 3.25;
areaVidrio = 2 * (anchura * altura);
```

En términos genéricos las sentencias que anteceden a estas dos líneas realizan el trabajo de obtener y almacenar los valores que posteriormente son procesados. Las sentencias que suceden a estas dos líneas realizan el trabajo de mostrar el resultado al usuario.

A continuación, vamos a analizar todas las líneas de nuestro programa para ver detalladamente la función que realizan cada una de ellas.

using System;

Esta instrucción le indica al compilador de C# que queremos utilizar elementos incluidos en el *espacio de nombres* `System`. Un espacio de nombres es un contenedor abstracto donde determinados nombres (denominados en ocasiones como identificadores) tienen un significado especial. Las personas también utilizamos espacios de nombres en nuestras conversaciones, si yo estoy utilizando el espacio de nombres "Fútbol" y digo que *"Ese equipo está 'on fire'"* estoy indicando algo bueno. Sin embargo, si yo estuviera utilizando el espacio de nombres "Bombero", yo estaría diciendo algo malo.

En el caso de C# el espacio de nombres `System` es en donde se describen un montón de elementos útiles que podemos utilizar en nuestros programas. Uno de esos elementos útiles proporcionados junto con C# es el objeto `Console` que nos permite escribir mensajes en la pantalla del usuario. Cuando quiera referirme a `Console`, tendré que comunicar al compilador que quiero utilizar el espacio de nombres `System`. Esto significa que, si hago referencia a algún elemento con un nombre concreto, el compilador buscará en el espacio de nombres `System` para ver si encuentra alguno que coincida con ese nombre. Utilizaremos otros espacios de nombres más adelante.

La clase CalcularAcristalamiento

Un programa C# se compone de una o más *clases*. Una clase es un contenedor que contiene datos y código para realizar un determinado trabajo en particular. En el caso de nuestra calculadora de doble acristalamiento, la clase sólo contiene un único método que calculará las longitudes de madera y área de vidrio necesarias para la fabricación de la ventana.

Es necesario otorgar un identificador a cada clase que construya. Yo he llamado a esta `CalcularAcristalamiento`, dado que este nombre refleja el trabajo que realiza la clase en este caso. Por ahora, no se preocupe demasiado por el tema de las clases; tan sólo asegúrese de utilizar nombres identificativos para las clases que construya.

Ah, y una cosa más. Hay una convención por la cual el nombre del archivo que contiene una clase en particular debe coincidir con el de la propia clase, en otras palabras, el código del programa anterior debe estar contenido en un archivo llamado `CalcularAcristalamiento`.cs.

static

Con esta palabra reservada se asegura que el método que sigue a esta palabra esté siempre presente, es decir, la palabra `static` en este contexto significa que "forma parte de la clase envolvente y siempre está aquí". Cuando empecemos a trabajar con objetos, nos daremos cuenta de que esta palabra reservada cuenta con todo tipo de repercusiones interesantes. Por ahora bastará con que se asegure de añadirla para que sus programas funcionen correctamente.

void

`void` significa "nada". En términos de programación la palabra reservada `void` significa que el método que creamos a continuación no retorna nada de interés para nosotros. El método realizará un trabajo y después finalizará. En algunos casos escribiremos métodos que devolverán un resultado (de hecho, luego en el programa vamos a utilizar un método de este tipo).

Sin embargo, para impedir que alguien accidentalmente pueda utilizar el valor retornado por nuestro método `Main`, estamos indicando explícitamente que el método no devuelve nada. Esto hace nuestro programa mucho más seguro, ya que el compilador sabe ahora que, si alguien intenta utilizar el valor devuelto por este método, esto debe ser un error.

Main

Como hemos indicado anteriormente debemos dar nombre a todos los métodos que vayamos creando siendo importante que sean descriptivos con el trabajo que éstos realizan. Esto es cierto, con excepción del método `Main`. A partir de este método (siempre debe de haber uno, y ser único) el programa se pone en funcionamiento. Cuando se carga y se ejecuta el programa, el primer método que toma el control es el método `Main`. Si se olvida de escribir el método `Main`, el sistema no sabrá por dónde empezar a ejecutar el programa.

()

Estos son un par de paréntesis que no encierran ningún tipo de información en su interior. Aunque esto pueda parecer estúpido, sirve para indicarle al compilador que el método principal no tiene parámetros. Un parámetro proporciona a un método algún dato o información con la que trabajar. Cuando define un método, puede decirle a C# que trabaje con una o más elementos, por ejemplo, `sin(x)` trabaja devolviendo el seno del ángulo especificado en `x`. Cubriremos el tema de los métodos con mayor grado de detalle un poco más adelante.

{

Esto es una *llave de apertura*. Las llaves en programación deben venir en pares de dos, es decir, toda llave de apertura debe tener su correspondiente llave de cierre. Las llaves permiten a los programadores agrupar trozos de código. Tales trozos de código reciben el nombre de *bloques*. Un bloque puede contener declaraciones de variables, seguidas posteriormente de una secuencia de instrucciones de programa que son ejecutadas en el orden en el que están escritas. En este caso las llaves delimitan las partes del trabajo que realiza el método principal *`Main`*.

Cuando el compilador encuentra la llave de cierre correspondiente al final del bloque, sabe que ha alcanzado el final del método y pasa a buscar otro (si lo hubiera). Los efectos de tener llaves desemparejadas son siempre fatales….

double

No, no se equivoque no estamos incitándole al programa a que se tome un trago. Esto no es lo que `double` significa en este contexto. Su significado en este caso es "*un número en punto flotante de precisión doble*".

Nuestro programa necesita recordar ciertos valores mientras se ejecuta. Cabe destacar que será necesario introducir los valores para la anchura y la altura de las ventanas, y posteriormente calcular e imprimir los valores para el área de vidrio y la longitud de la madera. Se denomina *variables* a los lugares en donde se guardan los valores. Al comienzo de cualquier bloque puede decirle a C# que desea reservar algún espacio en memoria para almacenar los valores de algunos datos. Cada información puede contener un determinado tipo de valor. Esencialmente, C# puede almacenar tres tipos de datos: números de punto flotante, números enteros y por supuesto cadenas de texto (es decir, letras, números y signos de puntuación). Al proceso de creación de una variable se le denomina *declaración* de la variable.

Puede declarar varias variables de un determinado tipo indicando el tipo de dato que almacenarán, seguida de la lista de nombres identificativos que desee otorgar a dichas variables. Por ahora, estamos utilizando el tipo `double`. Más adelante vamos a utilizar variables de otros tipos de datos.

anchura, altura, longitudMadera, areaVidrio

En C# una lista de elementos es separada por caracteres `,` (coma). En este caso, esta es una lista de nombres de variables. Una vez que el compilador ha visto la palabra reservada `double` (véase arriba) espera encontrar al menos el nombre identificativo de una variable para ser creada. El compilador trabaja con cada elemento de la lista, creando cajas con el suficiente espacio en memoria que puedan almacenar valores de tipo `double`, y dándoles a cada una de estas cajas en memoria el nombre que ha considerado apropiado. A partir de este instante, podemos hacer referencia a los nombres identificativos establecidos anteriormente, y el compilador sabrá que queremos utilizar esa variable en particular.

Punto del Programador: Conocer de dónde provienen los datos

En realidad, dadas las limitadas precisiones en las que podemos medir con una cinta métrica, y conociendo el hecho de que no vamos a poder fabricar ninguna ventana tan grande como el Universo, un número en punto flotante de precisión doble es excesivo para esta aplicación. Por tanto, tendría que preguntar a su cliente si está de acuerdo en expresar tan solamente las dimensiones en milímetros. Examinaremos las consideraciones que debemos tomar a la hora de escoger entre los distintos tipos de variables existentes un poco más adelante. Todas estas consideraciones son proporcionadas por los metadatos (datos sobre datos) que se reúnen y describen a la hora de construir el sistema.

;

El punto y coma marca el final del listado de los nombres identificativos de las variables, y también el final de esta instrucción de declaración. Todas las instrucciones en C# están separadas unas de otras por el carácter `;` (punto y coma), esto ayudar al compilador a ir por el buen camino.

El carácter `;` es muy importante. Este le dice al compilador dónde finaliza una instrucción. Si el compilador no encuentra uno de estos caracteres donde lo espera encontrar, mostrará un error en pantalla. Puede equiparar estos caracteres con las perforaciones existentes en los films de películas. Estos pequeños orificios rectangulares son los encargados de mantener la sincronización de toda la película.

string cadenaAnchura, cadenaAltura;

Anteriormente, hemos creado algunas variables que pueden almacenar números. En esta línea creamos otras dos variables que pueden almacenar cadenas de texto. Tenemos que crearlas de este

tipo porque cuando nosotros leemos dígitos numéricos introducidos por un usuario, primeramente, tenemos que leerlos en formatos de cadenas de texto. A continuación, convertiremos ese texto en formato numérico. Las variables `cadenaAnchura` y `cadenaAltura` (tenga en cuenta que los nombres identificativos de las variables son sensibles a minúsculas y mayúsculas) contendrán versiones de texto de los números que introduzcamos.

cadenaAnchura =

Esta es una sentencia de *asignación*. En esta sentencia, vamos a cambiar el valor de esta variable. Nuestro programa leerá una línea de texto introducida por el usuario, y guardará el resultado en la variable `cadenaAnchura`. Recuerde que una variable es tan sólo un espacio en memoria de un tamaño concreto, que puede almacenar un elemento de tipo simple (en este caso una cadena de texto).

Una buena parte de sus programas contendrán instrucciones que asignarán nuevos valores a las variables, así como a los diferentes resultados que desee calcular. En C# se utiliza el operador `=` para realizar asignaciones. La primera parte de esta sentencia es el nombre de la variable previamente declarada. A este le sigue el operador `=` (igual a) al que yo llamo el operador "da iguá", porque tengo la teoría de que su inventor, el galés Robert Recorde no lo creó porque no hay dos formas que puedan ser más iguales entre sí, sino porque no se le ocurría ningún otro nombre y cansado y de todo se preguntó ¿qué nombre le pongo? "Da iguá", no me como más la cabeza… y así se quedó. Perdónenme, no volveré a contar esto nunca más.

Console.

A la derecha del signo igual, tenemos el elemento que va a ser asignado a cadenaAnchura. En este caso se asignará la cadena devuelta por el método `ReadLine`. Este método es parte de un objeto llamado `Console` que es el encargado de controlar los flujos de entrada y salida de la consola. El punto (`.`) separa el identificador de objeto del identificador de método.

ReadLine

Indica que el método `ReadLine` ha de ser invocado. Esta acción pide al programa que se está ejecutando, que entre a este método, realice lo que las sentencias indiquen, y por último regresa al punto donde el programa fue interrumpido. Los métodos le permiten dividir su programa en varios fragmentos, pudiendo otorgar a cada uno de ellos un trabajo o tarea específica. Se puede decir que un método está formado por un fragmento de código reutilizable y del cual podemos hacer uso en cualquier momento deseado dentro de la aplicación. El sistema C# contiene una serie métodos construidos para realizar todo tipo de tareas en nuestros programas. `ReadLine` es uno de ellos.

Cuando el programa se ejecuta y el método `ReadLine` es invocado, éste espera a que el usuario introduzca una línea de texto y presione la tecla Intro. El método `ReadLine` devuelve una cadena de texto, que en este caso almacenamos en la variable `cadenaAnchura`.

()

Una llamada a un método es seguida a continuación por los *parámetros* del método. Un parámetro es un valor que se le pasa a un método para que realice alguna operación o tarea con él. Piense en éstos como las materias primas necesarias para realizar un algún tipo de proceso. `ReadLine` no necesita de materias primas; ya que éste sólo recoge la información introducida por el usuario a través del teclado. Sin embargo, TODAVÍA debemos proporcionar una lista de parámetros incluso aunque ésta se encuentre vacía.

;

Ya hemos visto el punto y coma anteriormente. Éste marca el final de una sentencia de nuestro programa.

anchura =

Esta es otra sentencia de asignación. A la variable `anchura` se le está dando un valor. La mayoría de las sentencias de los programas simplemente mueven y realizan asignaciones de datos.

double.

Quizás esta parte de la sentencia que estamos viendo pueda asustarle. No se preocupe ni se sienta intimidado, es más fácil de entender de lo que pueda parecerle. Estamos pidiendo al Sr. double (el responsable de lo concerniente a los números en punto flotante de precisión doble) que realice un trabajo para nosotros. En este caso el pequeño trabajo que tiene que realizar es “tomar la cadena almacenada en `cadenaAnchura` y convertirla en un número punto flotante de precisión doble.” El Sr. double proporciona esta capacidad exponiendo un método llamado `Parse`.

Tenga en cuenta que no hay nada erróneo o malo en que C# exponga sus métodos. Cuando tenga que diseñar programas más grandes, se dará cuenta de que la mejor manera de hacer esto es creando componentes que expongan los métodos para realizar el trabajo. El conjunto de bibliotecas de C# proporciona una serie de métodos que podemos importar o incluir en nuestros programas. Estos archivos contienen las especificaciones de diferentes funcionalidades ya construidas y utilizables que podremos agregar a nuestros programas. Una de las cosas que usted a las que tendrá que enfrentarse es a conocer dónde se encuentran los métodos y cómo utilizarlos. Como ocurre con muchas cosas en esta vida, el truco está en saber a quién preguntar…

Parse

El método `Parse` tiene la tarea de convertir la cadena que se le ha pasado en un número punto flotante de precisión doble equivalente. Para realizar este caso debe mirar dentro de la cadena (analizarla sintácticamente), extraer sucesivamente cada dígito correspondiente y luego calcular el valor real, así "12" significa un diez y dos unidades. Este proceso de analizar sintácticamente una cadena es denominado parseo. De ahí el nombre del método que estamos utilizando. El método toma la cadena que ha de ser parseada y devuelve el número equivalente de ésta que ha encontrado.

Tenga en cuenta que esto podría provocar errores terribles, porque si el usuario no teclea un valor del tipo esperado, la llamada al método `Parse` no sería capaz de resolver la operación debido a que espera un número y fallará al intentar realizar la operación que debe mostrar como resultado. Cómo se produce este error y cómo podemos prevenir en nuestro programa que este error no se produzca, quedará pendiente para una futura sección, así añadimos más emoción a este texto.

(cadenaAnchura);

Hemos visto que una llamada de un método debe ser seguida por las materias primas (*parámetros*) que necesita ese método. En el caso de `ReadLine` no hay parámetros, así que nosotros debemos suministrar una lista vacía para indicar esto. En el caso de `Parse` el método necesita recibir una cadena con la que trabajar. Nosotros hacemos esto indicando el nombre de la variable de cadena que contiene el texto (`cadenaAnchura`) dentro de un paréntesis como se muestra más arriba. El valor de la información en `cadenaAnchura` (es decir, el texto que el usuario ha escrito) es pasado al método `Parse` para que trabaje con ésta y extraiga el número almacenado en ella.

cadenaAltura = Console.ReadLine();

altura = double.Parse(cadenaAltura);

Estas dos sentencias repiten simplemente el proceso de leer en el texto introducido el valor de la altura, y a continuación almacenar el valor de la altura de la ventana convertida en punto flotante de precisión doble.

longitudMadera = 2(anchura + altura)3.25;

Esta es la parte realmente esencial del programa, ya que realiza el trabajo que muestra la información que requiere nuestro cliente. Esta sentencia toma los valores de altura y anchura y los utiliza para calcular la longitud de la madera requerida.

El cálculo se realiza con la sentencia señalada más arriba, siendo importante que tenga en cuenta los paréntesis utilizados en esta expresión. Normalmente C# resolverá las expresiones de la forma en que se puede esperar, es decir, se realizarán primero todas las multiplicaciones y divisiones,

seguidas por las sumas y restas. En la expresión anterior quiero realizar algunas operaciones primero, así que tal y como yo haría en matemáticas, utilizaré los paréntesis para darles prioridad a esas operaciones que tengo que calcular.

Tenga en cuenta que yo utilizo un factor de `3.25` por el hecho de que el cliente desea conocer la longitud de la madera necesaria para el doble acristalamiento en pies. Hay aproximadamente `3.25` pies en un metro, por eso yo multiplico el resultado en metros por este factor.

A los símbolos matemáticos `+` y `*` de la expresión se les denominan *operadores*, ya que indican que se debe llevar a cabo una operación especificada. A los otros elementos de la expresión se les denomina *operandos*, siendo éstos los argumentos o variables con los que los operadores trabajan.

areaVidrio = 2 * (anchura * altura);

Esta línea repite el cálculo para el área del vidrio. Tenga en cuenta que el área es dada en metros cuadrados, por lo que no se requiere ningún tipo de conversión. He indicado una multiplicación entre paréntesis que me permite indicar que estoy calculando dos veces el área (para dos hojas de vidrio). No hay necesidad de hacer esto particularmente, pero creo que esto lo hace más claro de ver.

Console.WriteLine

Esta es una llamada a un método, similar al método `ReadLine`, excepto que éste toma los parámetros que le hemos pasado y los imprime posteriormente por la consola.

(

En esta parte se inicia el envió de los *parámetros* que debe utilizar el método `WriteLine`. Anteriormente, también utilizamos parámetros en las llamadas a los métodos `Parse` y `ReadLine`

“La longitud de la madera es “

Esto es una cadena literal, una cadena de texto que será literalmente mostrada en pantalla por el programa. Las cadenas de texto son encerradas entre comillas dobles para indicarle al compilador que forman parte de una información real en el programa, y no instrucciones propias para el compilador.

+

El operador de suma (**+**) es un operador de adición. Ya hemos visto que éste se aplica para sumar dos números enteros. Sin embargo, en este caso significa algo totalmente diferente. En el caso que nos ocupa significa "concatena estas cadenas".

Tendrá que acostumbrarse al *contexto* en sus programas. Lo vimos anteriormente con los espacios de nombres. Ahora lo hemos vuelto a ver con los operadores. El sistema de C# utiliza el contexto de una operación para decidir qué hacer. En el caso del anterior **+**, posicionado entre dos números punto flotantes de precisión doble significa "realiza una suma". Aquí éste tiene una cadena a su lado izquierdo. Esto significa que va a realizar una concatenación y no una adición matemática.

longitudMadera

Este es otro ejemplo de contexto. Anteriormente hemos utilizado `longitudMadera` como representación numérica del valor de la longitud de la madera requerida. Sin embargo, en el contexto que se está utilizando en esta línea de código (añadida al final de la cadena) ésta no trabaja así.

El compilador C# debe pedir que se conviertan los datos del elemento `longitudMadera` en una cadena, para poder ser utilizado correctamente en esta situación. Afortunadamente se realiza correctamente, y el programa funciona tal como era de esperar.

Es muy importante que entienda exactamente lo que está ocurriendo aquí. Considere:

```
Console.WriteLine ( 2.0 + 3.0 );
```

Esta sentencia realiza un cálculo numérico (`2.0 + 3.0`) produciendo un valor punto flotante de precisión doble, mostrando la siguiente salida por pantalla:

```
5
```

Pero tenga en cuenta que la siguiente línea de código:

```
Console.WriteLine ( "2.0" + 3.0 );
```

Podría considerar que el operador `+` concatena dos cadenas. Sin embargo, en esta ocasión se pide previamente que el valor `3` sea convertido a una cadena (suena extraño – pero esto es lo que sucede). Esta sentencia imprime el siguiente resultado por pantalla:

```
2.03
```

La cadena "`2.0`" tiene el texto del valor `3.0` añadido al final de la cadena. Esta diferencia de comportamiento es ocasionada por el contexto de la operación que se está realizando.

Puede pensar en que todas las variables en nuestro programa son etiquetadas con *metadatos* (aquí está otra vez esa palabra), que el compilador utilizará para decidir qué hacer con éstas. La variable `cadenaAltura` está etiquetada con una información que dice "Esto es una cadena. Utilice un operador de adición junto a ésta, y obtendrá una concatenación". La variable `longitudMadera` está etiquetada con metadatos que dice "Esto es un valor en punto flotante de precisión doble. Utilice un operador de adición junto a éste, y obtendrá una operación aritmética".

+ " pies"

Aquí tenemos otra concatenación. En este caso añadimos la palabra "pies" al final de la cadena. Cuando se imprime un valor por pantalla, **siempre** hay que indicar al final la unidad con la que estamos trabajando. Esto le da mayor sentido a la sentencia.

)

El paréntesis de cierre indica el final de los parámetros que construimos para la llamada al método `WriteLine`. Cuando el método es llamado, el programa primero ensambla una cadena completa de todos los componentes, añadiendo (o concatenando) todos ellos para producir un único resultado. A continuación, pasará el valor de la cadena resultante al método, que lo imprimirá en la consola.

;

El punto y coma marca el final de esta sentencia.

}

Aquí aparece uno de los elementos realmente importantes. El programa está casi completado. Hemos añadido todos los procedimientos que necesitábamos. Sin embargo, todavía no hemos indicado al compilador que nuestro programa termina aquí. Esta primera llave de cierre señala el final del bloque de código perteneciente al cuerpo del método principal (`Main`). Un bloque de código comienza con `{` y finaliza con `}`. Cuando el compilador ve esta primera llave de cierre se dice a sí mismo "aquí finaliza el método principal (`Main`)".

}

La segunda llave de cierre realiza el mismo trabajo que la primera, pero en esta ocasión indica el final de la clase `CalcularAcristalamiento`. En C# todo se encuentra dentro de una clase. Una clase es un contenedor de propiedades y métodos. Si queremos (y nosotros haremos esto más adelante) podemos establecer un mayor número de métodos dentro de una clase. Por ahora, sin embargo, sólo necesitamos un método en esta clase. Así que utilizamos la segunda llave de cierre para marcar el final de la clase en sí.

Console.ReadKey

Esta es una llamada a un método. Este método obtiene la siguiente tecla de carácter o de función presionada por el usuario. En los casos que veremos en este libro, lo utilizamos en la parte final de nuestros programas, para poder visualizar el resultado de nuestro código, sin que se cierre la ventana CMD.

Signos de puntuación

Una de las cosas que habrá notado, es que utilizamos una gran cantidad de signos de puntuación en el código del programa. Éstos son de vital importancia y deben ser proporcionados exactamente tal y como C# los requiere, de lo contrario obtendrá lo que en programación denominamos un error de compilación. ¡Este error simplemente indica que el compilador es demasiado estúpido como para entender lo que le hemos indicado que haga!

Pronto se acostumbrará a detectar y cazar errores de compilación. Una de las cosas que descubrirá es que el compilador no siempre detecta el error en donde se ha cometido; considere el efecto de olvidar un carácter "`(`". Debe tener en cuenta que, aunque el compilador le indique que su código no contiene errores, ¡esto no le garantiza de que el programa haga lo que realmente espera!

Otra cosa que debe recordar es que el estilo que utilice para escribir el programa no afecta y no es determinante para el compilador, de hecho, el siguiente estilo es igual de válido que el anterior

```
using System;class CalcularAcristalamiento{static void Main(){double
anchura, altura, longitudMadera, areaVidrio;string cadenaAnchura,
cadenaAltura;cadenaAnchura = Console.ReadLine();anchura =
double.Parse(cadenaAnchura);cadenaAltura = Console.ReadLine();altura =
double.Parse(cadenaAltura);longitudMadera = 2 * ( anchura + altura ) *
3.25;areaVidrio = 2 * ( anchura * altura );Console.WriteLine ("La longitud
de la madera es " + longitudMadera + " pies" );Console.WriteLine("El área
del vidrio es " + areaVidrio + " metros cuadrados" );}}
```

- aunque si alguien escribe un programa representado de esta manera, ¡me encargaré de golpearle en los nudillos!

Punto del Programador: El estilo en el que se escriben los programas es muy importante

Se habrá dado cuenta del estilo tan atractivo que he utilizado para escribir el programa. Las líneas de código presentes en el interior de las llaves son indentadas para mejorar la legibilidad del código fuente, y así saber en qué nivel se encuentran dentro de todo el código. Por tanto, no hago esto por tener un listado de líneas de código con una estética artísticamente agradable. Lo hago porque de lo contrario leer los programas sería una tortura para cualquier programador y complicaría el comprender el trabajo que hace el programa. Aligualqueresultadifícilleeruntextosinespaciosdeseparación, encuentro demasiado duro leer el código fuente de un programa si éste no tiene un estilo correcto.

2.2 Manipulación de Datos

En esta sección vamos a echar un vistazo a cómo podemos escribir programas que manipulen datos, cómo podemos almacenar valores, recuperarlos y manipularlos. Esto nos proporciona la capacidad de realizar la parte del procesamiento de datos de los programas.

2.2.1 Variables y Datos

En el programa de la ventana de doble acristalamiento, hemos decidido almacenar la anchura y la altura de la ventana que estamos fabricando en dos variables que nosotros hemos descrito como de tipo `double`. Antes de que nosotros podamos avanzar en nuestra carrera por convertirnos en un buen programador, tenemos que considerar lo que esto significa, y conocer en qué otros tipos de datos podemos almacenar los datos de los programas que escribamos.

Los programas operan sobre datos. Un lenguaje de programación debe proporcionarle una forma de almacenar los datos que está procesando, de lo contrario no tendría sentido que pudiera procesar los datos. Lo que los datos signifiquen realmente, es una cuestión que usted como programador decide (véase la anterior digresión sobre datos).

Una variable es el nombre de una localización o espacio donde puede almacenar algo. Puede hacerse una idea de ésta, pensando en una caja de un tamaño determinado con un nombre pintado en ella. El nombre dado y pintado en la caja, le sirve para reflejar lo que va a ser almacenado dentro de ésta (en el programa anterior, nosotros utilizamos nombres identificativos como `longitudMadera`). También es necesario que elija el *tipo* de la variable (tamaño y forma de la caja) de entre todo el rango de tipos de almacenamiento que C# proporciona. El tipo de la variable es parte de los metadatos de esa variable.

Los programas también contienen *valores literales*. Un valor literal es tan sólo un valor en su programa que utiliza para algún propósito. Para cada tipo de variable, el lenguaje C# tiene una forma en que los valores literales de ese tipo se expresan.

2.2.2 Almacenamiento de números

Cuando consideramos valores numéricos, podemos encontrar dos tipos de datos:

- Números que representan valores simples, por ejemplo, el número de ovejas en un campo, los dientes de un engranaje, manzanas en una canasta. Éstos son conocidos como *enteros*.
- Números que no se pueden representar con valores simples, por ejemplo, la temperatura actual, la longitud de un trozo de cuerda, la velocidad de un automóvil. Éstos son conocidos como *reales*.

En el primer caso, siempre podemos almacenar el valor exacto; así que tendremos un número exacto de estos elementos. Estos son números *enteros*.

En el segundo caso, nunca podremos almacenar con exactitud lo que estamos viendo. Incluso si se mide un trozo de cuerda con 100 cifras decimales, no va a dar con su longitud exacta - podríamos siempre obtener un valor con mayor precisión. Estos son números *reales*. Una computadora es digital, es decir, opera enteramente sobre patrones de bits que pueden ser considerados como números. Debido a que nosotros sabemos que ésta trabaja en términos de on's and off's, ésta tiene problemas a la hora de almacenar valores reales. Para manejar valores reales, la computadora los tiene que almacenar en realidad con una exactitud limitada, que nosotros consideramos adecuada (y generalmente lo es).

Esto significa que cuando queremos guardar algo, tenemos que decirle a la computadora si se trata de un número entero o real. También tenemos que tener en cuenta el rango de valores posibles que tenemos que guardar para que podamos elegir el tipo adecuado a la hora de almacenar los datos.

Puede decirle a C# qué variable quiere crear declarándola. La declaración también identifica el tipo de elemento que queremos guardar. Piense en esto como que C# crea una caja de un tamaño determinado, diseñada específicamente para objetos del tipo especificado. La caja es etiquetada con algún *metadato* (aquí está otra vez esa palabra) para que el sistema sepa lo que puede guardar en el interior de la caja y de qué manera esa caja puede ser utilizada.

Almacenamiento de valores enteros

Los números enteros son los tipos más fáciles de almacenar para una computadora. Cada valor se corresponderá con un patrón particular de bits. El único inconveniente es de rango. Cuanto más grande sea el valor, mayor será el número de bits que necesitaremos para representarlo.

C# proporciona una gran variedad de tipos de enteros, que deberemos utilizar teniendo en cuenta previamente el rango de valores que deseemos almacenar:

sbyte	8 bits	de -128 a 127
byte	8 bits	de 0 a 255
short	16 bits	de -32768 a 32767
ushort	16 bits	de a 65535
int	32 bits	de -2147483648 a 2147483647
uint	32 bits	de 0 a 4294967295
long	64 bits	de -9223372036854775808 a 9223372036854775807
ulong	64 bits	de 0 a 18446744073709551615
char	16 bits	de 0 a 65535

El tipo entero estándar int, puede contener números alarmantemente grandes en C#, en el rango de -2,147,483,648 a 2,147,483,647. Si quiere almacenar enteros incluso más grandes que éstos (aunque no se me ocurre para que podría necesitarlo) tiene disponible el tipo long.

Un ejemplo de variable entera, podría ser la que mantiene un seguimiento de la cantidad de ovejas en un campo:

```
int numeroDeOvejas;
```

Este código crea una variable que podría hacer un seguimiento de más de 2000 millones de ovejas. Éste también permite a un programa manipular “ovejas en negativo” lo que probablemente no le resulte de utilidad. Recuerde que el lenguaje en sí, no es consciente de tales consideraciones. Si quiere asegurarse de que nunca va a tener más de 1.000 ovejas y que el número de ovejas nunca va a ser negativo, debe ser usted mismo el que agregue este comportamiento.

Al editar el código fuente de su programa utilizando Visual Studio (o cualquier otro editor de código que soporta *resaltado de sintaxis*) comprobará que los nombres de los tipos de datos soportados por el lenguaje C# (como int y float) se muestran en azul, tal como se muestra arriba.

Punto del Programador: Compruebe los cálculos matemáticos

Una cosa que siempre debe tener en cuenta, es que un programa no siempre detectará cuando se ha excedido el rango de una variable. Si establezco el valor 255 en una variable de tipo byte, esta operación es correcta y totalmente válida, puesto que 255 es el mayor valor posible que una variable de este tipo puede contener. Sin embargo, si añado uno al valor de esta variable, el sistema puede que no detecte esta operación como un error. De hecho, esto puede causar que el valor "de la vuelta" a 0 (consulte aritmética de saturación), lo que podría causar grandes problemas a mi programa.

Valores literales enteros

Un literal entero se expresa como una secuencia de dígitos sin punto decimal:

```
23
```

Este es el valor entero 23. Podría usarlo en mi programa de la siguiente manera:

```
numeroDeOvejas = 23;
```

Si estoy utilizando uno de los tipos más "cortos", el valor literal es considerado por el compilador como de ese tipo:

```
sbyte peqValor = 127;
```

En esta declaración, el 127 es considerado como un literal sbyte, no como un entero. Esto significa que, si hago algo estúpido como, por ejemplo:

```
sbyte peqVal = 128;
```

(el valor máximo que puede almacenar un sbyte es 127) el compilador detectará que he cometido un error y el programa no compilará.

Almacenar valores reales

"Real" es un término genérico utilizado para describir y expresar aquellos números que no forman parte del conjunto de los números enteros. Estos números tienen un punto decimal y una parte fraccionaria. Dependiendo del valor decimal, el punto flota a lo largo del número, de ahí el nombre del tipo, `float` (flotante).

C# proporciona un tipo de caja que puede almacenar un número real. Un valor `float` (flotante) estándar tiene un rango de entre `1.5E-45` y `3.4E48` con una precisión de tan solamente 7 dígitos (es decir, no tan buena como la mayoría de las calculadoras de bolsillo).

Si desea una mayor precisión (sus programas utilizarán más memoria y se ejecutarán más lentamente), puede utilizar una doble caja en su lugar (`double` es la abreviatura de precisión doble). Como hemos indicado anteriormente, este tipo `double` ocupa más memoria de la computadora, pero tiene un rango de entre `5.0E-324` y `1.7E308` con una precisión de 15 dígitos.

Un ejemplo de utilización de una variable `float`, podría ser para almacenar el precio promedio de un helado:

```
float precioPromedioHeladoEnPeniques;
```

Un ejemplo de utilización de una variable `double`, podría ser para almacenar la anchura del universo en pulgadas:

```
double anchuraUniversoEnPulgadas;
```

Finalmente, si quiere lo último en precisión puede usar el tipo `decimal`. Este tipo utiliza el doble de espacio de almacenamiento que la del tipo `double` y almacena valores con una precisión de 28-29 dígitos. Se utiliza en los cálculos financieros donde los números no son tan grandes, pero tienen que ser calculados y almacenados a muy alta precisión.

```
decimal robSaldoDeudor;
```

valores literales reales

Hay dos formas en las que se pueden almacenar números de punto flotante; como `float` o como `double`. Cuando introduce valores literales dentro del programa, al compilador le gusta saber si está escribiendo un valor de punto flotante (caja de tamaño más pequeño) o de precisión doble (caja de tamaño grande).

Un literal `float`, se expresa al igual que expresaría un número real terminado, pero con terminación en `f`:

```
2.5f
```

Un literal `double`, se expresa como un número real, pero sin la `f`:

```
3.5
```

También puede usar exponentes para expresar los valores dobles y flotantes:

```
9.4605284E15
```

Este es un valor literal de precisión doble, destinado en el ejemplo anterior para almacenar el número de metros en un año luz. Si introduce una `f` en el extremo final, se convierte en un valor literal de punto flotante.

A diferencia de la forma en que trabaja con los números enteros, el compilador es bastante quisquilloso con los números reales y sobre la forma en que éstos pueden o no ser combinados. Esto es debido a que cuando traspasa un valor de una variable de doble precisión a una variable de punto flotante pierde algo de precisión. Esto significa que tiene que tomar medidas especiales para asegurarse que como programador tiene claro que desea que esto suceda y que asumirá las consecuencias. Este proceso es conocido como *casting* y lo veremos en detalle un poco más tarde.

Punto del Programador: Los tipos básicos son los más apropiados

Apreciará que yo, y la mayoría de programadores, tendemos a utilizar solamente variables de tipo entero (int) y punto flotante (float). Esto puede parecer un excesivo despilfarro (es muy poco probable que alguna vez necesite hacer un seguimiento de dos mil millones de ovejas), pero de esta forma los programas son más fáciles de entender.

2.2.3 Almacenar Texto

A veces la información que queremos almacenar es texto. Este texto puede estar formado a veces por un único carácter; y en otras ocasiones por una cadena de caracteres. C# proporciona variables para trabajar con ambos tipos de datos:

Variables char

Una variable de tipo `char` representa una instancia de un solo carácter. Un carácter es lo que se obtiene cuando se pulsa una tecla en un teclado, o la muestra de un único carácter en pantalla. C# utiliza un conjunto de caracteres llamado UNICODE que puede manejar más de 65,000 diseños de caracteres diferentes, incluyendo una amplia gama de alfabetos distintos.
Un ejemplo de utilización de una variable de tipo `char`, podría ser para almacenar la tecla de comando que el usuario acaba de presionar:

```
char teclaDeComando;
```

valores literales char

Puede expresar un carácter encerrándolo entre comillas simples:

```
'A'
```

Esto significa "el carácter A", y es lo que su programa obtendría si le pidiera que leyera un carácter del teclado, y el usuario mantuviera presionada la tecla shift y pulsara al mismo tiempo la tecla A. Si está desarrollando su programa utilizando un editor que soporta *resaltado de sintaxis*, comprobará que un carácter literal se muestra en color rojo.

Caracteres de secuencia de escape

Con relación a lo explicado en el apartado anterior, hay a una interesante pregunta que probablemente se esté haciendo: "¿Cómo expresamos entonces el carácter ' (comilla simple)?". Esto se logra mediante el uso de una secuencia de escape. Esta es una secuencia de caracteres que comienzan con un carácter especial de escape. Un *carácter de escape* es un carácter individual que suprime cualquier significado especial que tenga el carácter o secuencia de caracteres que le sigue. Las secuencias de escape deben de estar precedidas por el carácter de escape \ (barra diagonal inversa). Las secuencias de escape disponibles son:

Carácter asociado	Nombre de la secuencia de escape
\'	Comilla simple
\”	Comillas dobles
\\	Barra diagonal inversa
\0	Null
\a	Campana (alerta)
\b	Retroceso
\f	Avance de página
\n	Nueva línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical

El efecto de estas secuencias de escape depende del dispositivo al que los esté enviando. Algunos sistemas emiten un sonido cuando les envía el carácter Campana (alerta). Algunos otros limpian la pantalla cuando les envía el carácter Avance de página.

Puede utilizarlos como sigue:

```
char campana = ‘\a’;
```

Tenga en cuenta que la a debe estar en minúsculas.

Valores de codificación de caracteres

Hemos establecido que la computadora realmente manipula números en lugar de letras. C# utiliza el estándar Unicode para convertir los caracteres a los números que los representan. Si lo desea, puede expresar un carácter literal como un valor del conjunto de caracteres Unicode. La buena noticia es que esto le da acceso a una gran variedad de caracteres (tan amplio como sea su conocimiento de las codificaciones representativas). La mala noticia es que debe expresar este valor en hexadecimal, que es un sistema un poco más complejo de utilizar que el decimal. La mejor noticia de todas es que probablemente no necesite hacer este tipo de tareas tan a menudo.

A modo de ejemplo, el valor Unicode de la a mayúscula (A) es 65. Se representa en formato hexadecimal (base 16) como 41. Por tanto, puedo escribir la siguiente instrucción en mi programa:

```
char Amayuscula = '\x0041';
```

Tenga en cuenta que he tenido que poner ceros a la izquierda de los dos dígitos hexadecimales.

Variables *string*

Un tipo de caja que puede contener una cadena de texto. En C# una cadena puede ser muy corta, por ejemplo "Rob", pero también puede ser muy larga, por ejemplo "Guerra y Paz" (que además de tres palabras, es el título de un libro). Una variable de tipo `string` puede contener una línea de texto. Sin embargo, debido a que hay un carácter especial que significa "nueva línea" (véase más arriba), es perfectamente posible que una simple cadena contenga un gran número de líneas de texto.

Un ejemplo de utilización de una variable de tipo string, podría ser para almacenar justamente lo que el usuario acaba de teclear:

```
string lineaDeComandos;
```

valores literales *string*

Un valor literal `string` se expresa entre comillas dobles:

```
"esto es una cadena"
```

La cadena puede incluir las secuencias de escape vistas previamente:

```
"\x0041BCDE\a"
```

Si imprimimos la cadena por pantalla, se mostraría lo siguiente:

```
ABCDE
```
- *y el sistema tratará de emitir el sonido Campana (alerta).*

Si solamente estamos expresando texto sin caracteres de escape, podemos indicarle al compilador que esto es una cadena *literal*. Lo hacemos poniendo @ delante del literal:

```
@"\x0041BCDE\a"
```

Si imprimimos esto por pantalla, obtendríamos la siguiente salida:

```
\x0041BCDE\a
```

Esto puede ser útil cuando se está expresando rutas de archivos. El carácter de cadena literal también puede utilizarse para obtener literales de cadena multilíneas:

```
@"El veloz

zorro marrón

salta sobre el perro perezoso"
```

Esto expresa una cadena que se extiende sobre tres líneas. Estos saltos de línea en la cadena se conservan cuando son almacenados en una variable.

2.2.4 Almacenando Valores de Estado utilizando Booleanos

Una variable de tipo `bool` (abreviatura de booleano), es un tipo de caja que puede almacenar alguna información sea o no verdadera. Si, por ejemplo, está almacenando la información de si un usuario ha pagado o no por una suscripción a un servicio, no hay ninguna necesidad de gastar espacio en memoria mediante el uso de un tipo que pueda contener un gran número de valores posibles. En lugar de ello, bastará con tan solo guardar los estados `true` o `false`. Estos son los dos únicos valores que permite el tipo booleano.

Otro ejemplo de utilización de una variable de tipo `bool`, podría ser para almacenar el estado de una conexión de red:

```
bool estadoRedOK;
```

valores literales *bool*

Los valores de tipo booleano se expresan fácilmente mediante sus dos únicos estados posibles, `true` o `false`:

```
estadoRedOK = true;
```

Punto del Programador: Piense en el tipo de sus variables

Elegir correctamente del tipo que debe ser una variable es como un arte. Personalmente, tiendo a utilizar el almacenamiento de punto flotante solamente como último recurso, ya que generalmente no me interesa almacenar datos con tanta precisión. Teniendo un poco de ingenio se puede trabajar con números enteros, que ofrecen mayor simplicidad. Por ejemplo, en lugar de almacenar el precio de un artículo como 1.5 libras (necesitando el uso de una variable de tipo `float`), puede almacenarlo como 150 peniques.

Además, cuando se considera la forma de almacenar los datos es importante recordar de dónde provienen éstos. En el ejemplo anterior, he utilizado el tipo `double` para guardar la anchura y la altura de una ventana. Esto es **realmente inútil**, ya que no tiene en cuenta la precisión requerida o la precisión con la que se puede medir la ventana. Me imagino que el vendedor de vidrio no será capaz de medir con una precisión superior a 1 mm, por lo que no tendría sentido almacenar los datos para ese tipo de precisión.

Este caso, ejemplifica otra consideración sobre los metadatos. Cuando esté ofreciendo la posibilidad de guardar la información en punto flotante; averigüe cómo se está produciendo el tipo de información con la que va a trabajar, antes de decidir cómo se almacenará. Como ejemplo, puede pensar que, para almacenar y trabajar con la velocidad de un coche, es necesario crear una variable punto flotante. Sin embargo, cuando averigua que el sensor de velocidad, solamente ofrece una precisión para calcular el desplazamiento por cada milla por hora, esto hace el trabajo mucho más sencillo.

2.2.5 Identificadores

En C#, un identificador es un nombre que elige el programador para asignar a un elemento de un programa. Comúnmente, el nombre que asignamos a una variable es conocido como *identificador*. No obstante, veremos otros lugares en los programas en donde nosotros también creamos identificadores. Debe tener en cuenta que todo identificador debe cumplir las siguientes reglas de sintaxis:

- Todos los nombres de identificadores deben comenzar por una letra o carácter de subrayado "_".
- A continuación de esa primera letra o carácter de subrayado, los nombres de identificadores pueden tener más letras, números o más caracteres de subrayado "_".
- Los nombres de identificadores, no deben tener espacios intermedios. También hay que recordar que las vocales acentuadas y la eñe son problemáticas, porque no son letras "estándar" en todos los idiomas, así que no son válidas como parte de un identificador.

También debe tener en cuenta que las letras mayúsculas y minúsculas se consideran distintas, por tanto, `Fred` y `fred` son considerados diferentes identificadores.

A continuación, se muestran algunos ejemplos de declaraciones, una de las cuales no es válida (a ver si es capaz de adivinar cuál es y por qué):

```
int fred;

float jim;

char 29yosoy;
```

Una de las reglas de oro de la programación, junto con la de "*utiliza siempre el teclado con las teclas hacia arriba*" es:

Siempre debe dar a sus variables nombres significativos.

De acuerdo con lo publicado en las novelas románticas de la editorial Mills & Boon, las mejores relaciones son aquellas que son significativas.

La convención en C# para dar nombre a las variables que tengamos que crear, es combinar letras mayúsculas y minúsculas, teniendo en cuenta que cada palabra del identificador debe comenzar con una letra mayúscula, excepto la primera de ellas:

```
float precioPromedioHeladoEnPeniques;
```

Este estilo de escritura, es denominado *camel case*. El nombre se debe a que las letras iniciales de cada palabra se escriben en mayúsculas, asemejándose a las jorobas de un camello.

Punto del Programador: Piense en el nombre de sus variables

La elección de los nombres de las variables es otra habilidad que debe desarrollar. Estos nombres deben ser lo suficientemente largos para ser identificativos, pero no tan largos como para hacer complejas las líneas de código de su programa. Tal vez el nombre `precioPromedioHeladoEnPeniques` pueda ser considerado excesivo si tenemos en cuenta este último punto. De cualquier forma, creo que podría vivir con él y aceptarlo. Recuerde que siempre puede hacer cosas razonablemente coherentes con su estilo de diseño de programación para que el código de su programa se vea bonito y sea claramente entendible:

precioPromedioHeladoEnPeniques =
precioTotalComputadoEnPeniques / numeroDeHelados;

2.2.6 Asignar valores a las variables

Una vez que hemos conseguido declarar una variable, ahora tenemos que saber cómo guardar algo en ella, para así posteriormente poder obtener el valor almacenado cuando lo necesitemos. C# realiza el proceso de asignar valores por medio de una sentencia de asignación.

Hay dos partes en una asignación, el elemento que quiere asignar y el destino (lugar) donde desea establecerlo, por ejemplo, considere el siguiente código:

```
class Asignacion
{
      static void Main()
      {
            int primero, segundo, tercero;
            primero = 1;
            segundo = 2;
            tercero = segundo + primero;
            Console.ReadLine();
      }

}
```

Código de Ejemplo 02 Ejemplo de Asignación absurda

La primera parte del programa le debe parecer bastante familiar por ahora. Dentro de la función `Main`, hemos declarado tres variables, `primero`, `segundo` y `tercero`. Todas ellas son de tipo entero.

Las tres últimas sentencias son las que realmente hacen el trabajo. Estas sentencias son sentencias de asignación. Una asignación otorga un valor a una variable concreta, teniendo en cuenta que debe ser del tipo apropiado para almacenar dicho valor (sea consciente siempre de esto, puesto que, al compilador, como ya sabemos, ni le interesa ni sabe, lo que usted está realmente tratando de hacer). El valor que es asignado es una expresión. El signo igual del medio no debería confundirle, éste no significa que sean iguales en sentido numérico; significa que el resultado de la operación de la derecha será almacenado en el campo de la izquierda:

```
2 = segundo + 1;
```

- es una instrucción de asignación peligrosa que haría aparecer toda clase de errores.

Expresiones

Una expresión es algo que puede ser evaluado para producir un resultado. Una vez obtenido el resultado, podemos utilizarlo en nuestro programa para lo que necesitemos. Las expresiones

pueden ser tan sencillas como un valor simple, y tan complejas como una operación de cálculo. Las expresiones están compuestas por dos elementos, *operandos* y *operadores*.

Operandos

Los operandos son los elementos con los que los operadores trabajan. Por lo general son valores literales o identificadores de variables. En el programa anterior `primero`, `segundo` y `tercero` son identificadores, mientras que `2` es un valor literal. Un valor literal es un elemento que está literalmente ahí en el código. Un valor literal tiene un tipo asociado a éste por el compilador.

Operadores

Los operadores son los elementos que realizan el trabajo: En ellos se especifica la operación a realizar sobre los operandos. La mayoría de los operadores trabajan sobre dos operandos, uno a cada lado. En el programa anterior + es el único operador.

A continuación, algunos ejemplos de expresiones:

```
2 + 3 * 4
-1 + 3
(2 + 3) * 4
```

Estas expresiones son evaluadas y resueltas por C# de izquierda a derecha, tal como lo haría usted mismo. Incluso, al igual que ocurre en las matemáticas tradicionales todas las multiplicaciones y divisiones en una expresión son realizadas en primer lugar, seguidas de las operaciones de suma y resta.

C# realiza este trabajo dando a cada operador una prioridad. Cuando C# evalúa una expresión revisa todos los operadores que tiene con la más alta prioridad y realiza estas operaciones en primer lugar. A continuación, busca los operadores que tienen el siguiente nivel de prioridad para de nuevo resolver sus operaciones, y así continua sucesivamente hasta que obtenga el resultado final de la expresión. Tenga en cuenta que esto significa que la primera expresión indicada arriba, devolverá como resultado `14` y no `20`.

Si quiere forzar el orden en que las operaciones se realizan, puede encerrar entre paréntesis a los operandos que sean necesarios, como se muestra al final del ejemplo anterior. Incluso puede encerrar entre paréntesis a operandos que ya se encuentren previamente encerrados entre paréntesis, siempre que se asegure de que tiene el mismo número de ellos tantos de apertura como de cierre. Al ser un alma sencilla, siempre tiendo a hacer las cosas muy claras, encerrando todo entre paréntesis.

Probablemente no valga la pena preocuparse por esta expresión de evaluación, como la gente pija llama a esta; generalmente las cosas se resuelven tal y como se espera.

Para complementar la información anterior, a continuación, se muestra una lista de operadores, que incluye la descripción de lo que éstos hacen y su precedencia (prioridad). Este listado presenta los operadores ordenados por su orden de prioridad: los operadores con prioridad más alta, primero.

-	operador unario negativo, es utilizado en C# para representar números negativos, por ejemplo, -1. El operador unario negativo se aplica a un solo elemento.
*	multiplicación, tenga en cuenta que se utiliza el carácter * para no confundirlo con el carácter x utilizado matemáticamente.
/	división, debido a la dificultad de dibujar un número encima de otro en una pantalla, usamos este carácter en su lugar.
+	suma.
-	resta. Tenga en cuenta que éste es exactamente el mismo carácter que utilizamos para representar números negativos.

Esta lista no recoge todos los operadores disponibles en C#, pero sí los operadores que utilizaremos por ahora. Debido a que estos operadores trabajan con números, a menudo son denominados como operadores numéricos. Pero, por supuesto, debe recordar que algunos de ellos (por ejemplo, el operador **+**) también puede ser utilizado entre otros tipos de datos. También es posible utilizar operadores para convertir valores de un tipo a otro. Esto puede causar problemas como veremos a continuación.

2.2.7 Conversiones de Tipos de Datos

Siempre que yo convierto el valor de un tipo a otro, el compilador de C# se muestra muy interesado en lo que estoy haciendo. Éste se preocupa de si la operación que estoy a punto de realizar, provocará una pérdida de información o no. A la conversión que se realiza de forma automática sin peligro de pérdida de información, se la denomina "*conversión implícita*". A continuación, nos referimos a ella como "*Ampliación*". A la conversión con peligro de pérdida de datos, se la denomina "*conversión explicita*". A continuación, nos referimos a ella como "*Reducción*".

Ampliación (Conversión Implícita) y Reducción (Conversión Explícita)

El principio general que C# utiliza es que, si está "reduciendo" un valor, siempre se le pida explícitamente qué es lo que quiere hacer. Si está ampliando un valor, no tendrá problemas.

Para entender lo que significan estos términos, vamos a utilizar maletas. He decidido irme de viaje, así que, para llevar el equipaje he tomado una maleta. Si decidiera cambiar la maleta escogida por otra más pequeña, tendría que sacar todo lo que he metido en la maleta más grande e introducirlo

en la más pequeña. Pero podría ocurrir que no tuviese espacio, así que tendría que sacar una de mis camisas. Esto es un ejemplo de "reducción" (también llamado a veces “truncamiento”).

Sin embargo, si cambiase mi maleta por una más grande, no tendría ningún problema. En la maleta más grande podré meter todo lo que hay en la maleta pequeña, y me sobrará espacio.

En términos de C#, el "tamaño" de un tipo es el rango de valores (el mayor y el menor) y la precisión (el número de decimales), que pueda tener. Esto significa que si escribo:

```
int i = 1;
float x = i;
```

Esto funciona bien porque el tipo punto flotante puede almacenar todos los valores soportados por el tipo entero. Sin embargo:

```
float x = 1;
int i = x;
```

- causaría que el compilador protestara (incluso aunque por el momento la variable `x`, solamente almacena un valor entero). Al compilador le preocupa la pérdida de información que pueda tener, por lo que tratará esta asignación como si fuese un error.

Tenga en cuenta que esto se aplica dentro de los valores de punto flotante, así como, por ejemplo:

```
double d = 1.5;
float f = d;
```

- provocaría también un error, dado que el compilador sabe que una variable de tipo `double` tiene el doble de precisión que otra variable de tipo `float`.

Casting (Conversión Explícita)

Podemos forzar a C# a considerar un valor como si fuese de cierto tipo mediante el uso del proceso de *casting*. Este proceso obliga al compilador a considerar un valor como si fuese de un tipo particular. Para realizar una conversión explícita, debemos especificar entre paréntesis el tipo al que vamos a aplicar dicha conversión delante del valor o la variable que vamos a convertir. Ejemplo:

```
double d = 1.5;
```

```
float f = (float) d;
```

En el código anterior el mensaje al compilador es "Estoy al tanto de que esta asignación puede producir una pérdida de datos. Me aseguraré que mi programa funcione correctamente". Puede considerar al proceso de casting como una forma en la que el compilador se lava las manos ante un eventual problema tras realizar esta acción. Así que recuerde, si un programa fallara a causa de la perdida de información, no es porque el compilador hiciera algo mal.

Como hemos visto anteriormente, cada tipo de variable tiene un intervalo particular de valores posibles, y el rango de valores de punto flotante es mucho más amplio que para los enteros. Esto significa que si hace asignaciones como esta:

```
int i;
i = (int) 123456781234567890.999;
```

- la operación de conversión de tipo está condenada al fracaso. El valor que se asigna a `i` en el código anterior, no será válido. C# no comprobará errores de este tipo. Asegúrese siempre de que su programa nunca excederá el rango de los tipos de datos que está utilizando – ¡el programa no lo tendrá en cuenta, pero el usuario sin lugar a dudas sí lo tendrá!

Un proceso de casting también puede provocar perdida de información de otras formas:

```
int i;
i = (int) 1.999;
```

El código anterior toma el valor 1.999 (que se compila como un valor de tipo `double`) y lo convierte a tipo `int`. Este proceso descarta la parte fraccionaria, lo que significa que la variable `i`, terminará teniendo valor 1, aun cuando el número original estuviese mucho más cercano al 2. Debe recordar que este truncamiento, se lleva a cabo siempre que realice el cast de un valor con una parte fraccionaria (float, double, decimal) a otro valor sin parte fraccional.

Casting y Valores Literales

Hemos visto que podemos expresar valores "literales" dentro de nuestros programas. Estos son solamente valores que desea utilizar en sus cálculos. Por ejemplo, en nuestro programa de doble acristalamiento tuvimos que multiplicar la longitud de la madera en metros por 3.25 para convertir el valor de la longitud de metros a pies (hay alrededor de 3.25 pies en un metro).

C# realiza un seguimiento de los elementos que está uniendo, y esto incluye la forma en que permite que los valores literales se utilicen. Esto significa que sentencias como:

```
int i;
i = 3.4 / "estúpidez";
```

- serán tratadas con el desprecio que merecen. El compilador de C# sabe que es un disparate dividir el valor `3.4` entre la cadena "`estúpidez`".

Sin embargo, considere esto:

```
float x;
x = 3.4;
```

Este código parece legítimo. Sin embargo, no lo es. Esto es debido a que el valor literal `3.4` es un valor de doble precisión cuando se expresa como un literal, y la variable `x` ha sido declarada como punto flotante. Si quiero establecer un valor literal punto flotante en una variable de tipo punto flotante, puedo hacer uso del casting:

```
float x;
x = (float) 3.4;
```

Este código convierte el literal de doble precisión en un valor de punto flotante, de modo que la asignación es correcta y funcional.

Para hacer la vida más sencilla, los creadores de C# han añadido una manera diferente en la que podemos expresar un valor literal de punto flotante en un programa. Si escribe una `f` después del valor, éste será considerado como un valor de punto flotante. Esto significa que:

```
float x;
x = 3.4f;
```

- compilará correctamente.

2.2.8 Tipos de Datos en Expresiones

Cuando C# utiliza un operador, toma una decisión en cuanto al tipo del resultado que se va a producir. Básicamente, si los dos operandos son enteros se dice que el resultado debe ser un número

entero. Si los dos sonoperandos son de tipo punto flotante se dice que el resultado debe ser de punto flotante. Esto puede ocasionar problemas, considere lo siguiente:

```
1/2
1/2.0
```

Podría pensar que ambas expresiones darían el mismo resultado. No es así. El compilador piensa que la primera expresión, que involucra solamente a números enteros, debería dar un resultado de tipo entero. Así pues, el valor de esta operación daría como resultado `0` (la parte fraccionaria siempre se trunca). En la segunda expresión, al estar involucrada en ella un valor punto flotante, será evaluada para ofrecer un resultado punto de flotante de doble precisión. En este caso `0.5`.

La forma en que se comporta un operador depende de su contexto. Más adelante veremos que el operador `+`, que normalmente realiza un cálculo numérico, puede ser utilizado entre cadenas para concatenarlas, es decir, "`Ro`" + "`b`" dará como resultado "`Rob`".

Si desea tener un control completo sobre el tipo particular de operador que el compilador generará, el programa debe contener conversiones explícitas para establecer el contexto adecuado para el operador.

```
using System;
class DemoCasting
{
	static void Main()
	{
		int i = 3, j = 2;
		float fraccion;
		fraccion = (float) i / (float) j;
		Console.WriteLine("fracción : " + fraccion);
		Console.ReadKey();
	}

}
```

Código de Ejemplo 03 Ejemplo de Casting

El operador cast de conversión de tipo (`float`), solicita al compilador que considere los valores almacenados en las variables enteras como si fuesen de tipo punto flotantes, por lo que obtendremos un resultado por pantalla de `1.5`, en lugar de `1`.

Punto del Programador: El proceso de casting proporciona claridad

Tiendo a establecer instrucciones casting, incluso en casos en que no son necesarias. Probablemente no afecte al resultado del cálculo, pero ofrece información al lector de lo que estoy tratando de hacer, y logra que el programa sea más fácil de leer.

2.2.9 Programas y Patrones

En este punto podemos volver a nuestro programa de doble acristalamiento y echar un vistazo en la manera en la que trabaja. El código que realmente hace el trabajo se reduce a sólo unas pocas líneas de código que leen los datos, almacenan sus valores en un tipo apropiado de ubicación y posteriormente se utilizan los valores almacenados para calcular el resultado que el usuario requiere:

```
string cadenaAnchura = Console.ReadLine();
double anchura = double.Parse(cadenaAnchura);

string cadenaAltura = Console.ReadLine();
int altura = double.Parse(cadenaAltura);

longitudMadera = 2 * ( anchura + altura ) * 3.25;

areaVidrio = 2 * ( anchura * altura );

Console.WriteLine ( "La longitud de la madera es " +
      longitudMadera + " pies" ) ;
Console.WriteLine ( "El área del vidrio es " +
      areaVidrio + " metros cuadrados" );
```

Lo interesante es que este patrón de comportamiento se puede reutilizar una y otra vez. Como ejemplo, considere otro amigo suyo, que tiene una farmacia. Él quiere un programa que calcule el costo total de los comprimidos que compra, y el número de botes que necesita. Él introducirá el costo de los comprimidos y el número que requiere. Los comprimidos siempre se venden en botellas que pueden contener hasta 100 comprimidos.

Usted puede modificar fácilmente su programa para que realice este trabajo, la parte más difícil es calcular cuántas botellas se necesitan para un determinado número de comprimidos. Si realiza una división dividiendo un determinado número entero de comprimidos entre 100, obtendrá resultados erróneos (para cualquier número de comprimidos menor de 100, su programa indicará que son necesarios 0 botes). Una manera de resolver esto es añadir 99 al número de comprimidos antes de

realizar la división, forzando el número de botellas requeridas para "redondear hacía arriba" cualquier número de comprimidos mayor que 0. El código funcional sería como sigue:

```
int cantidadBotes = ((cantidadComprimidos + 99) / 100);
int precioVenta = cantidadBotes * precioPorBote;
```

Podemos completar el código del programa de la siguiente manera:

```
string cadenaPrecioPorBote = Console.ReadLine();
int precioPorBote = int.Parse(cadenaPrecioPorBote);

string cadenaCantidadComprimidos = Console.ReadLine();
int precioPorBote = int.Parse(cadenaCantidadComprimidos);
int cantidadBotes = ((cantidadComprimidos + 99) / 100);

int precioVenta = cantidadBotes * precioPorBote;

Console.WriteLine ( "El número de botes necesarios es: " +
                                    cantidadBotes );
Console.WriteLine ( "El precio total es: " +
                                    precioVenta );
```

Lo interesante en este este punto, es que el programa para el químico es en realidad una simple variación del programa para el vendedor de doble acristalamiento. Ambos se ajustan a un patrón de comportamiento (leer los datos de entrada, procesarlos, mostrarlos por pantalla) que son muy comunes en muchas aplicaciones. Cualquier programa donde se requiera leer algunos datos, realizar alguna operación con ellos y mostrar el resultado en pantalla podrá hacer uso de este patrón.

Un buen programador debe saber identificar el mejor patrón para resolver la naturaleza de un problema.

2.3 Escribir un programa

Los programas que hemos creado hasta ahora han sido muy simples. Solamente leen algunos datos, realizan alguna operación con esos datos y muestran el resultado en pantalla. Sin embargo, vamos a necesitar crear programas que hagan cosas más complejas que estas. Los programas que realicemos probablemente tengan que tomar una serie de decisiones basándose en los datos que le hemos ofrecido. También podrían tener la necesidad de repetir ciertas acciones hasta que se cumplan una condición. Y por supuesto, también es posible que tengan que leer una gran cantidad de datos y luego procesarlos en un gran número de formas diferentes.

En esta sección vamos a considerar cómo podemos dar a nuestros programas ese nivel adicional de complejidad, y contemplar a nivel general el negocio de desarrollar programas.

2.3.1 Software como una historia

Algunas personas afirman que escribir un programa es similar a escribir una novela. Yo no estoy completamente convencido de que esto sea cierto. Es verdad, que a veces he sido consciente de que algunos manuales de computación son como obras de ficción, pero los programas son algo más que eso. Creo que, si bien lo que hacemos no puede decirse que sea una novela como tal, es indiscutible que un buen programa debe tener algunas características asociadas a la buena literatura:

- Debe ser fácil de leer. En ningún momento el lector debe verse forzado a retroceder o poner al día sus conocimientos que el escritor asume que tiene. Todos los nombres deben tener sentido y ser distintos unos de otros.

- Debe tener una buena puntuación y gramática. Los diversos componentes deben organizarse de una manera clara y coherente.

- Debe verse bien sobre la página. El código de un programa debe estar bien estructurado. Los diferentes bloques deben estar indentados y las sentencias distribuidas sobre la página de una manera coherente y fundamentada.

- Debe quedar claro quien lo escribió, y cuando fue modificado por última vez. Si escribe algo bueno usted debe poner su nombre en él. Si modifica lo que ha escrito, debe añadir información sobre los cambios realizados y las razones que le han llevado a hacerlo.

Una gran parte fundamental de un programa bien escrito, se basa en los comentarios que el programador escriba dentro de éste. Un programa sin comentarios es como un avión que cuenta con piloto automático, pero que no tiene ventanas. Existe la posibilidad de que le lleve al lugar correcto, pero será muy difícil saber hacía donde se dirige desde su interior.

Comentarios en bloque

Cuando el compilador de C# ve la secuencia "`/*`", que significa el inicio de un comentario, se dice a sí mismo:

*"¡Ajá! Aquí hay un fragmento de información para que mentes pensantes superiores a la mía la tengan en cuenta. Voy a ignorar todo lo que sigue hasta que vea un ***/** que cierre el comentario."*

Como ejemplo:

```
/* Este programa calcula la cantidad de vidrio y madera requerida para
   realizar una ventana de doble acristalamiento. */
```

Sea generoso con sus comentarios, ya que éstos ayudan a hacer su programa mucho más fácil de entender. Se sorprenderá mucho al descubrir lo rápido que se olvida cómo logro conseguir que su programa funcionara correctamente. También puede utilizar los comentarios para informar sobre la versión actual del programa, cuando fue modificado por última vez y por qué, y el nombre del programador que lo escribió – aunque fuese usted mismo.

Si está utilizando un editor que soporta *resaltado de sintaxis* se dará cuenta que los comentarios se muestran normalmente en color verde.

Comentarios de línea

Otra forma de realizar comentarios es utilizando la secuencia `//`. Esto marca el comienzo de un comentario que se extiende hasta el final de la línea de código en particular. Es útil para poner realizar un apunte, observación o indicación tras una sentencia:

```
posicion = posicion + 1; // pasar al siguiente cliente
```

He comentado la sentencia anterior para dar información extra sobre lo que hace.

Punto del Programador: No añada demasiados detalles

Escribir comentarios es algo muy sensato de hacer. Pero no se vuelva loco. Recuerde que la persona que está leyendo su código probablemente conozca el lenguaje C# y no necesita que le explique ciertas cosas con demasiado detalle:

```
contadorCabras = contadorCabras + 1;  // añadir una Cabra
```

Esto es un evidente insulto al lector. Si utiliza identificadores evidentes, puede esperar que éstos expliquen mejor lo que hace ese trozo de código que un simple comentario.

2.3.2 Control de flujo del programa

Controlar el *flujo* es determinar el orden en el que se ejecutarán las instrucciones que tenemos en nuestros *programas*. Nuestro primer programa de doble acristalamiento es muy simple, ya que se ejecuta directamente desde la primera declaración a la última, y luego se detiene. Frecuentemente se encontrará con situaciones en las que el programa debe decidir lo que tiene que hacer dependiendo de ciertos factores. Básicamente, podemos decir que hay tres tipos de flujo:

1. flujo secuencial: ejecución de instrucciones una tras otra
2. flujo selectivo: elegir dependiendo de una condición dada
3. flujo repetitivo: repetir de acuerdo a una condición dada

Cada programa está compuesto normalmente de los tres tipos vistos arriba, ¡y poco más! Este listado puede resultarle útil para obtener una visión global a nivel de diseño de cómo el programa va a funcionar. Hasta ahora sólo hemos considerado programas que se ejecutan en flujo secuencial. El camino que sigue un programa es llamado en ocasiones "hilo de ejecución". Cuando llama a un método el hilo de ejecución es transferido a ese método hasta que haya terminado.

Instrucción Condicional - if

El programa que procesa la madera y el vidrio destinado para nuestro amigo vendedor de ventanas de doble acristalamiento es válido; de hecho, nuestro cliente probablemente se encontraría bastante satisfecho con él. Sin embargo, no es perfecto. El problema no está tanto en el programa, sino en la forma en la que podría utilizarlo el usuario.

Si se introduce en el programa una anchura de ventana con valor -1, éste lo acepta y en consecuencia muestra un resultado no esperado. Nuestro programa no realiza ningún tipo de comprobación para las entradas de las anchuras y alturas. El usuario podría tener motivos de queja si el programa no reconoce que le ha dado un valor que no se esperaba, de hecho, actualmente se está librando una serie de casos en los tribunales de los Estados Unidos, donde un programa no ha podido reconocer datos no válidos, produciendo conflictos y causando mucho daño.

Lo que tendremos que hacer en este tipo de casos es informar al usuario que ha realizado una acción que no entra dentro de la lógica. En la especificación de nuestro programa, que damos al cliente, debemos decir algo como esto (en nuestros metadatos):

El programa rechazará dimensiones de ventana fuera de los intervalos siguientes:

```
anchura inferior a 0,5 metros
anchura superior a 5.0 metros
altura inferior a 0.75 metros
altura superior a 3.0 metros
```

Nosotros ya no podemos hacer nada más; Si el programa obtiene un valor para la anchura de 1 en lugar de 10 será un problema del usuario, ¡lo importante desde nuestro punto de vista es que la especificación anterior evita que nos demanden!

Para realizar esto, el programa debe detectar los valores incorrectos y rechazarlos, por lo que podemos utilizar la siguiente construcción:

```
if (condicion)
      sentencia o bloque a realizar si la condición se cumple (true)
else
      sentencia o bloque a realizar si la condición no se cumple (false)
```

La condición determina lo que ocurre en el programa. Así pues, ¿qué entendemos nosotros por una condición? C# tiene una manera en que `true` y `false` pueden ser expresados explícitamente en un programa. Ya hemos visto que el tipo `bool` se utiliza para almacenar estos estados *lógicos*.

Por tanto, podemos crear condiciones que devuelvan un resultado lógico. Estas son llamadas "condiciones lógicas". Lo cual es lógico. La condición más simple es el valor `true` o `false`, Por ejemplo:

```
if (true)
      Console.WriteLine ( "Hola, mamá" );
```

Esta condición es válida, aunque bastante inútil ya que la condición siempre va a ser verdadera, por lo que "hola mamá" siempre va a ser impreso en pantalla (hay que señalar que en este caso no hemos hecho uso de la parte `else` – y esto es correcto, ya que esa parte es opcional).

Operadores Relacionales en Instrucciones Condicionales

Para realizar tareas condicionales necesitamos un conjunto de operadores relacionales adicionales que podemos utilizar en las expresiones lógicas. Los operadores relacionales también trabajan con operandos, al igual que ocurre con los operadores numéricos. Sin embargo, una expresión implica que sólo puede producir uno de dos valores, `true` o `false`. Los operadores relacionales disponibles son los siguientes:

==

igual(es). Si el lado izquierdo y el lado derecho son iguales la expresión tiene el valor `true`. Si no son iguales el valor es `false`.

```
4 == 5
```

- da como resultado `false`. Tenga en cuenta que este operador no es particularmente válido para comparar variables de punto flotante, y comprobar si almacenan exactamente los mismos valores. Debido al hecho de que se almacenan con una precisión limitada, observará que las condiciones fallan cuando no deberían hacerlo, por ejemplo, en la siguiente ecuación:

```
x = 3.0 * (1.0 / 3.0);
```

- podría dar como resultado que x contuviera `0.99999999`, lo que significaría que:

```
x == 1.0
```

- resultaría **`false`** — incluso aunque matemáticamente la comprobación debería ser evaluada como **`true`**.

!=

distintos. Lo contrario al operador relacional *iguales*. Si los operandos no son iguales la expresión tiene el valor `true`, si son iguales tiene el valor `false`. Una vez más, esta comprobación no es conveniente para el uso con números de punto flotante.

<

menor que. Si el operando de la izquierda es menor que el de la derecha el valor de la expresión es `true`. Si el operando de la izquierda es mayor o igual que el de la derecha la expresión da como resultado `false`. Es válido comparar números de punto flotante de esta manera.

>

mayor que. Si el operando de la izquierda es mayor que el de la derecha el resultado es `true`. Si el operando de la izquierda es menor que o igual que el de la derecha el resultado es `false`.

<=

menor que o igual a. Si el operando de la izquierda es menor que o igual que el de la derecha se obtiene `true`, de lo contrario se obtiene `false`.

>=

mayor que o igual a. Si el operando de la izquierda es mayor que o igual que el de la derecha se obtiene `true`, de lo contrario se obtiene `false`.

!

not. Puede utilizarse para invertir un valor o expresión particular, por ejemplo, puede expresar !`true`, que es `false`, o puede expresar: !(x==y) – que significa lo mismo que (x!=y). No se utiliza cuando se desea invertir el sentido de una expresión.

Combinar Operadores Lógicos

En ocasiones queremos combinar expresiones lógicas, para hacer elecciones más complejas, por ejemplo, para comprobar la validez de la anchura de la ventana debemos comprobar que el valor sea superior al mínimo permitido y menor al máximo permitido. C# proporciona operadores adicionales para combinar valores lógicos:

&&

y. Si los operandos a cada lado de las condiciones && son `true`, el resultado es `true`. Si uno de ellos es `false` el resultado es `false`, por ejemplo

```
(anchura >= 0.5) && (anchura <= 5.0)
```

- esto sería true si la anchura fuera **válida** de acuerdo con nuestra descripción anterior. Encerrar entre paréntesis cada una de las condiciones, ayuda a ver más fácilmente lo que estamos haciendo. Sin embargo, el compilador es capaz de averiguar que el operador && necesita ser aplicado entre el resultado de las dos expresiones lógicas, por lo que no son realmente necesarios.

||

o. Si cualquiera de los operandos a cada lado de la || son `true`, el resultado de la expresión es `true`. La expresión solamente es `false` si ambos operandos son `false`, por ejemplo:

```
anchura < 0.5 || anchura > 5.0
```

- esto sería true si la anchura fuera **incorrecta**. Hablamos de que la anchura no es válida (incorrecta), si ésta es inferior al mínimo permitido o superior al máximo permitido. Tenga en cuenta que para invertir el sentido de la condición (es decir, true cuando el valor no es válido) no sólo tenemos que cambiar > por <= en cada expresión, sino también cambiar && por ||.

Utilizando estos operadores junto con la construcción `if`, podemos tomar decisiones y cambiar lo que nuestro programa va a hacer en respuesta a los datos que obtenemos.

Punto del Programador: Descomponga sus expresiones

Si es consciente que está construyendo condiciones enormes y complejas para lograr que una sentencia se cumpla, probablemente esto sea un síntoma de que realmente podría y debería utilizar más de una sentencia `if`, e indentar el código para clarificarla. Esto hará que el código sea más fácil de depurar, ya que podrá ver y analizar cada una de las condiciones utilizadas en el programa de una sola vez, en lugar de tener que descomponer una enorme condición en el orden correcto.

Utilizar Bloques para Combinar Sentencias

Hemos decidido que, si el usuario ingresa un valor fuera de nuestro rango permitido, se genere un error y el valor se establezca al máximo o mínimo correspondiente aceptado. Para realizar esto, tenemos que hacer dos declaraciones que son ejecutadas dentro de una condición particular, una para mostrar el mensaje por pantalla y la otra para llevar a cabo una asignación. Para esta tarea podemos utilizar los caracteres { y }. Una serie de sentencias agrupadas entre caracteres { y }, son consideradas como una sola instrucción, así que hacemos lo siguiente:

```
if ( anchura > 5.0 )
{
      Console.WriteLine (“Anchura excesiva, utilizando la máxima\n”);
      anchura = 5.0;
}
```

Las dos sentencias se encontrarán ahora dentro de un bloque, y solamente se ejecutarán si la anchura es superior a `5.0`. Puede agrupar cientos de sentencias de esta forma, que al compilador no le va a importar. También puede poner dichos bloques de código dentro de otros bloques, lo que en programación es denominado *anidamiento*.

El número de caracteres { y } en su programa deben ser coincidentes y por lo tanto parejos, ¡de lo contrario obtendrá errores extraños e incomprensibles, cuando el compilador alcance el final del programa a la mitad de un bloque o llegue al final de su programa a la mitad de su archivo!

Punto del Programador: Haga uso del Diseño para Informar al Lector

Hago que todo sea mucho más fácil de entender aumentando la indentación de cada bloque de código en un determinado número de espacios. Es decir, cada vez que abro un bloque de código con el carácter `{`, sitúo el margen izquierdo un poco más hacía la derecha. Con esto soy capaz en todo momento con un simple vistazo de saber en el nivel donde me encuentro dentro del código. Esto es algo que hacen los buenos programadores.

Metadatos, Números mágicos y constantes

Un número mágico es un valor con un significado especial. Su valor nunca cambiará a lo largo del programa; es una constante que se utiliza dentro de éste. Cuando desarrolle mi programa de doble acristalamiento, incluiré algunos números mágicos que corresponderán a los valores máximo y mínimo de alturas y anchuras. Estos valores máximos y mínimo, provienen de los metadatos que recopilé cuando escribí la especificación del programa.

En realidad, podría utilizar simplemente y directamente los valores `0.5`, `5.0`, `0.75` y `3.0` – pero por sí solos no tienen ningún significado especial, y hace que el programa sea más complejo de modificar. Si, por alguna razón, el tamaño máximo válido para el vidrio cambiara a `4.5` metros, tendría que revisar todo el programa y modificar solamente los valores correspondientes afectados por este cambio. Por esta razón, no soy partidario de utilizar simples "números mágicos" en los programas. Así que, en estos casos, lo que se hace es proporcionar un nombre significativo a cada valor literal numérico (“número mágico”) que necesitemos utilizar en un programa.

Podemos hacer esto mediante una variable constante, es decir, una variable que nunca debería cambiar de valor en el programa.

```
const double PI = 3.141592654;
```

Tras haber realizado una asignación como la anterior, podríamos a continuación hacer lo siguiente:

```
circulo = 2 * PI * radianes;
```

La asignación anterior hace al número más identificativo (estamos utilizando PI, no un valor anónimo), y además hace que el programa sea más rápido de entender, escribir y modificar. En cualquier parte del programa donde vaya a usar un número mágico, deberá utilizar una constante de la forma que hemos visto, por ejemplo:

```
const double MAX_ANCHURA = 5.0;
```

Existe una convención de escribir en LETRAS MAYÚSCULAS los nombres de las constantes. De esta forma, aumentamos la legibilidad en los programas, pudiendo observar de un simple vistazo los elementos definidos que nunca cambiarán tras la compilación del programa.

Podemos modificar nuestro programa de doble acristalamiento de la siguiente manera:

```
using System;
class CalcularAcristalamiento
{
      static void Main()
      {
            double anchura, altura, longitudMadera, areaVidrio;

            const double MAX_ANCHURA = 5.0;
            const double MIN_ANCHURA = 0.5;
            const double MAX_ALTURA = 3.0;
            const double MIN_ALTURA = 0.75;

            string cadenaAnchura, cadenaAltura;

            Console.Write ("Introduzca la anchura de la ventana: ");
            cadenaAnchura = Console.ReadLine();
            anchura = double.Parse(cadenaAnchura);

            if (anchura < MIN_ANCHURA) {
                  Console.WriteLine ("Anchura demasiado pequeña.\n\n");
                  Console.WriteLine ("Usando la mínima permitida");
                  anchura = MIN_ANCHURA;
            }

            if (anchura > MAX_ANCHURA) {
                   Console.WriteLine ("Anchura demasiado grande.\n\n") ;
                   Console.WriteLine ("Usando la máxima permitida");
                   anchura = MAX_ANCHURA ;
            }

            Console.Write ("Introduzca la altura de la ventana: ");
            cadenaAltura = Console.ReadLine();
            altura = double.Parse(cadenaAltura);

            if (altura < MIN_ALTURA) {
                   Console.WriteLine("\nAltura demasiado pequeña.");
                   Console.WriteLine ("Usando la mínima permitida");
                   altura = MIN_ALTURA;
            }

            if (altura > MAX_ALTURA) {
                   Console.WriteLine("\nAltura demasiado grande.\n\n");
```

```
                    Console.WriteLine ("Usando la máxima permitida");
                    altura = MAX_ALTURA;
                }

                longitudMadera = 2 * ( anchura + altura ) * 3.25;

                areaVidrio = 2 * ( anchura * altura );

                Console.WriteLine ("\nLa longitud de la madera es de " +
                    longitudMadera + " pies");
                Console.WriteLine("El área de vidrio es de " +
                    areaVidrio + " metros cuadrados");
                Console.ReadKey();
            }
        }
```

Código de Ejemplo 04 Calcular Acristalamiento Completo

Este programa cumple con nuestros requisitos. Esta nueva versión no usará valores incompatibles con nuestra especificación. Sin embargo, todavía no podría decirse que esta versión sea perfecta. Si nuestro vendedor introduce una altura errónea, el programa simplemente limita el valor y que hay que volver a ejecutarlo, provocando que tenga que introducir de nuevo la altura.

Lo que realmente queremos es una manera de poder pedir repetidamente valores de anchura y altura, hasta que sea introducido uno que se adapte a la especificación. C# nos permite hacer esto proporcionando construcciones en bucle.

2.3.3 Bucles

Las sentencias condicionales le permiten hacer algo si una determinada condición es verdadera. Sin embargo, con frecuencia querrá repetir algún proceso mientras se cumpla una determinada condición, o bien repetir algún proceso un número determinado de veces.

En C# tenemos tres maneras de hacer esto, dependiendo precisamente de lo que estemos tratando de realizar. Tenga en cuenta que contamos con tres métodos, no porque necesitemos utilizar los tres, sino para tener diversas opciones que nos harán la vida más fácil a la hora de escribir el programa (como un accesorio para nuestra motosierra que nos permite realizar una tarea en particular más fácilmente). La mayor parte de la habilidad de la programación consiste en elegir la herramienta o accesorio adecuado para realizar el trabajo que tenemos entre manos (¡el resto es averiguar por qué la herramienta no hizo lo que esperaba de ella!).

En el caso de nuestro programa, queremos pedir repetidamente números mientras no se introduzca ninguno válido, es decir, al recibir un número permitido nuestro bucle debe finalizar. Esto significa

que, si nosotros obtenemos el número correcto en el primer intento, el bucle se ejecutará una sóla vez. Podría pensar que voy demasiado rápido; lo único que he hecho es este cambio:

```
Obtener valores hasta que uno de ellos sea VÁLIDO
```

por

```
Obtener valores mientras ninguno de ellos sea VÁLIDO
```

Parte del arte de la programación consiste en cambiar la forma en la que está pensando el problema, adaptándose a las distintas maneras en la que pueda utilizar el lenguaje de programación para resolverlo.

bucle do -- while

En el caso de nuestro pequeño programa, utilizamos la construcción `do – while`, que tiene una estructura como esta:

```
do
      sentencia o bloque
while (condición)
```

Con esto podemos repetir un bloque de código hasta que la condición finalmente sea falsa. Tenga en cuenta que la comprobación se realiza después de haber ejecutado la instrucción o bloque, es decir, incluso si la comprobación está predestinada a fallar, la sentencia se realiza una vez.

Una condición en este contexto exactamente la misma que en una construcción `if`, otorgando la intrigante posibilidad de realizar programas como el siguiente:

```
using System;

class BucleInfinito
{
      public static void Main()
      {
            do
                  Console.WriteLine( "Hola, mamá" );
            while ( true );
      }
 }
```

Código de Ejemplo 05 Bucle Infinito

Este es un programa perfectamente válido. ¿Cuánto tiempo estará ejecutándose? Esta es una pregunta muy interesante. Para responder a esta cuestión tendríamos que tener nociones de

psicología humana, energías futuras y cosmología. Con toda probabilidad, éste se ejecutará hasta que:

1. Usted se aburra.
2. La energía eléctrica de su hogar se acabe.
3. El Universo implosione.

Esta es una situación dramática. Así cómo es posible cortar su pierna con una vieja motosierra si lo intenta tenazmente, es posible utilizar cualquier lenguaje de programación para escribir un programa que nunca finalice. Me recuerda a las instrucciones de uso de mi champú favorito:

1. Moje su pelo.
2. Añada champú y masajee suavemente hasta conseguir una buena espuma.
3. Enjuague su pelo con agua tibia.
4. Repita el proceso.

Me pregunto, ¿cuántas personas estarán todavía lavándose el pelo en este momento?

bucle while

En ocasiones querrá decidir si repetir o no el bucle antes de ejecutarlo. El bucle anterior se ejecuta después de que el código que ha de repetirse se haya realizado al menos una vez. Para nuestro programa, queremos pedir un valor al usuario, y poder decidir entonces si es o no es válido. Con el fin de ser lo más flexible posible, C# nos proporciona otra forma de construir el bucle, que nos permite realizar antes que nada la comprobación:

```
while
      (condición)
      sentencia o bloque de código
```

Tenga en cuenta que, al omitir la palabra reservada `do`, C# reduce el número de teclas que debe presionar para ejecutar el programa (¡si incluye la palabra reservada `do`, el compilador tendrá la enorme deferencia de mostrarle un mensaje de error – ¡pero estoy seguro que usted ya había supuesto esto!).

bucle for

A menudo tendrá que repetir un proceso un número determinado de veces. Las estructuras de control iterativas o de repetición, permiten realizar este tipo de trabajos con bastante facilidad:

```
using System;

class BucleWhileActuandoComoBucleFor
{
	public static void Main()
	{
		int i;
		i = 1;
		while ( i < 11 );
		{
			Console.WriteLine ( "Hola, mamá" );
			i = i + 1;
		}
		Console.ReadKey();

	}
}
```

Código de Ejemplo 06 Bucle While Actuando como Bucle For

Este inútil programa muestra por pantalla el saludo a nuestra madre 10 veces. Este proceso se realiza mediante el uso de una variable que controla el bucle. La variable especificada tiene un valor inicial (1), y es testeada cada vez que vuelve al inicio del bucle. El valor de la variable de control es incrementado en cada iteración. Eventualmente la variable alcanzará el valor 11, momento en el que la estructura de control finalizará y nuestro programa se detendrá.

C# proporciona una construcción para permitir establecer un bucle como el anterior en formato de una sola línea:

```
for ( inicialización ; condición ; actualización )
{
	tarea que nosotros queramos hacer
	un número determinado de veces
}
```

Podríamos utilizar esta construcción para reescribir el programa anterior de la siguiente forma:

```
using System;

class BucleFor
{
	public static void Main()
	{
```

```
            int i;
            for ( i = 1 ; i < 11 ; i = i + 1 );
            {
                  Console.WriteLine( "Hola, mamá" );
            }
            Console.ReadKey();
      }
}
```

Código de Ejemplo 07 Bucle For

Al inicio del bucle se establece el valor con el que la variable de control será inicializada. La instrucción condicional debe cumplirse para que el bucle `for` `--` continúe ejecutándose. La sentencia de actualización se lleva a cabo actualizando la variable de control en la parte final del bucle. Tenga en cuenta que los tres elementos están separados mediante el carácter punto y coma. La secuencia de eventos es la siguiente:

1. Establecer el valor inicial para la variable de control.
2. Comprobar si la condición se cumple y salir de la estructura de repetición si esto es así.
3. Si la condición no se ha cumplido, ejecutar el boque de código dentro de la estructura de repetición.
4. Realizar la actualización de la variable de control.
5. Repetir el proceso a partir del segundo punto.

Crear un bucle de este tipo es más rápido y sencillo que utilizar un bucle `while`, ya que éste mantiene todos los elementos del bucle en una misma línea de código. Utilizando esta forma, es menos probable que se olvide de otorgar a la variable de control su valor inicial, o de actualizarla.

Si usted se arma un lío con el valor de la variable de control del bucle, su programa realizará cosas imprevistas, por ejemplo, si vuelve a establecer el valor de `i` a `0` dentro del bucle, éste se ejecutará indefinidamente. Se lo tiene bien merecido.

Punto del Programador: No sea inteligentemente estúpido

Algunas personas, para demostrar lo inteligentes que son (léase con ironía), intentan realizar otro tipo de acciones astutas desde las sentencias de inicialización, condición y actualización, ya que consideran que pueden realizar otras acciones más complejas que no sean tan simples como las de asignar, incrementar y comprobar. Algunos programadores piensan que ellos parecerán más inteligentes, si pueden hacer todo el trabajo de la parte "interna" en la parte superior, dejando una sentencia vacía tras ella.

Yo denomino a esta clase de gente "personas estúpidas". Rara vez se necesita crear códigos complejos y enrevesados. Cuando esté desarrollando programas, las dos cosas por las que debe preocuparse son: "¿Cómo puedo comprobar que todo funciona correctamente?" y "¿Cómo de fácil es este código de entender?". El código complejo y enrevesado, no le ayuda a que se cumpla ninguno de estos dos puntos clave.

Interrumpir la ejecución de Bucles

A veces querrá salir de un bucle mientras se encuentra ejecutándose. Es decir, puede decidir que su programa no tiene necesidad de continuar ejecutando el bucle; o que no tiene un punto a donde ir, y desea salir del bucle para continuar con la ejecución del programa desde la instrucción posterior al bucle.

Puede hacer esto utilizando la instrucción `break`. Esta instrucción ordena al programa, abandonar el bucle inmediatamente. Su programa tomará la decisión de salir de esta forma. Esto es de mucha utilidad, porque nos permite proporcionar una opción en medio de alguna tarea de decir "¡abandona este punto!". Por ejemplo, en el siguiente fragmento de programa, la variable `abandona`, normalmente `false` pasará a ser `true` cuando el bucle sea abandonado; y la variable `seEjecuteCorrectamente`, normalmente `true`, pasará a ser `false` cuando finalice con normalidad.

```
while (seEjecuteCorrectamente)
{
      instrucciones complejas
      ....
      if (abandona)
      {
            break;
      }
      ....
      más instrucciones complejas
```

```
        ....
}

....
parte del código que alcanzamos si abandona es true
....
```

Tenga en cuenta que estamos utilizando dos variables como interruptores, éstas no almacenan valores como tal; ya que en realidad se utilizan para representar estados dentro del programa que se está ejecutando. Este es un truco de programación estándar que probablemente encontrará muy útil.

Puede escapar de cualquiera de los tres tipos de bucle. En todos los casos, el programa continúa ejecutándose desde la instrucción posterior al bucle, si existe alguna.

Punto del Programador: ¡Cuidado con las instrucciones de ruptura!

La palabra reservada `break` puede parecer similar a la instrucción `goto`, que tanto respeto impone entre los programadores. La estructura `goto` permite realizar saltos en la ejecución del código, desde el punto de ejecución en curso hasta una etiqueta predefinida. Por esta razón, la estructura `goto` es considerada como un recurso potencialmente peligroso y confuso. La instrucción `break` permite saltar desde cualquier punto de un bucle hasta la instrucción justo fuera del bucle. Esto significa que, si mi programa alcanza la sentencia inmediatamente después del bucle, hay un número determinado de maneras en las que podría haber llegado allí; una por cada `break` incluido en el bucle anterior a esta instrucción. Esto puede hacer que el código sea más difícil de entender. La construcción `break` es menos confusa que la construcción `goto`, pero igualmente puede acarrear problemas. Es por ello que, le aconsejo que tenga cuidado al usarla.

Regresar a la parte inicial de un bucle

De vez en cuando, querrá volver a la parte superior de un bucle y ejecutar todas las instrucciones de nuevo. Esto ocurre cuando el bucle ha alcanzado la instrucción que usted necesitaba. C# proporciona la palabra reservada `continue`, que indica algo similar a lo siguiente:

Por favor, no ejecutes ninguna de las instrucciones restantes de la iteración actual. Regresa al inicio del bucle, evalúa de nuevo la expresión condicional y realiza las actualizaciones oportunas, si es necesario.

En el siguiente programa, la variable booleana `Hacer_Esta_Vez_Todo_Lo_Que_Necesitamos` es establecida a `true` cuando haya llegado tan lejos como nosotros necesitamos.

```
for ( elemento = 1 ; elemento < Total_Elementos ; elemento=elemento + 1 )
{
      ....
      elementos que procesan tareas
      ....
      if (Hacer_Esta_Vez_Todo_Lo_Que_Necesitamos) continue;
      ....
      elementos adicionales que procesan tareas
      ....
}
```

La instrucción `continue` hace que el programa vuelva a recorrer el bucle si es necesario hacerlo. Puede considerarlo como un avance al paso 2 en el listado anterior.

Decisiones más complejas

Ahora podemos pensar en utilizar una estructura de control, para comprobar la validez de la altura y la anchura de una ventana. Básicamente, queremos mantener la petición de valores al usuario hasta que obtengamos uno que sea VÁLIDO; es decir, si obtenemos un valor que es superior al máximo permitido o inferior al mínimo permitido, pediremos otro. Para hacer esto, tenemos que realizar dos comprobaciones, para verificar que el valor introducido es VÁLIDO. Nuestro bucle debe continuar ejecutándose si:

```
anchura > MAX_ANCHURA o anchura < MIN_ANCHURA
```

Para realizar esta comprobación, utilizamos uno de los operadores lógicos descritos anteriormente para escribir una expresión que será cierta si la anchura no es válida:

```
if ( anchura < MIN_ANCHURA || anchura > MAX_ANCHURA ) ..
```

Punto del Programador: Acostúmbrese a revertir las condiciones

Una de las particularidades a la que se tendrá que acostumbrar es que a menudo tendrá que revertir la forma en las que ve las cosas. En lugar de decir "Léeme un número válido", tendrá que decir "Léeme números, mientras no sean válidos". Lo que significa que a menudo tendrá que realizar comprobaciones de acciones que no quiere que se cumplan.

Programa Acristalamiento completo

Esta es una solución completa al problema, que utiliza todos los detalles discutidos anteriormente.

```
using System;

class CalcularAcristalamiento
{
      static void Main()
      {
            double anchura, altura, longitudMadera, areaVidrio;

            const double MAX_ANCHURA = 5.0;
            const double MIN_ANCHURA = 0.5;
            const double MAX_ALTURA = 3.0;
            const double MIN_ALTURA = 0.75;
            string cadenaAnchura, cadenaAltura;

            do {
            Console.Write ("Introduzca anchura de la ventana entre " +
                  MIN_ANCHURA + " y " + MAX_ALTURA + ": ");
            cadenaAnchura = Console.ReadLine();
            anchura = double.Parse(cadenaAnchura);
            } while (anchura < MIN_ANCHURA || anchura > MAX_ANCHURA );

            do {
            Console.Write ("Introduzca altura de la ventana entre " +
                  MIN_ALTURA + " y " + MAX_ALTURA + ": ");
            cadenaAltura = Console.ReadLine();
            altura = double.Parse(cadenaAltura);
            } while (altura < MIN_ALTURA || altura > MAX_ALTURA );

            longitudMadera = 2 * ( anchura + altura ) * 3.25;

            areaVidrio = 2 * ( anchura * altura );

            Console.WriteLine ("La longitud de la madera es de " +
                  longitudMadera + " pies" ) ;
            Console.WriteLine("El área de vidrio es de " +
                  areaVidrio + " metros cuadrados");
            Console.ReadKey();
      }
 }
```

Código de Ejemplo 08 Calcular Acristalamiento con Bucles

2.3.4 Operadores abreviados de incremento y decremento

Hasta ahora, hemos visto operadores que trabajan con dos operandos, ejemplo:

```
contador_ventanas = contador_ventanas + 1
```

En este caso, el operador es el signo `+` y sus operandos son la variable `contador_ventanas` y el valor `1`. El propósito de la sentencia anterior es añadir `1` a la variable `contador_ventanas`. Sin embargo, esto es una manera bastante larga de expresarlo, tanto en términos de lo que tenemos que escribir como de lo que la computadora realmente hace cuando ejecuta esta instrucción en el programa. C# nos permite ser más concisos si lo deseamos, la línea:

```
contador_ventanas++
```

`-` realizaría la misma asignación que la anteriormente vista. Expresándonos de esta manera más concisa, el compilador puede generar código más eficiente, ya que éste sabe que lo que queremos hacer es añadir `1` a una variable en particular. El operador `++` es llamado *operador unario*, porque trabaja solamente con un operando. Éste hace que el valor de este operando sea aumentado en uno. Existe también un operador `–` equivalente, que puede utilizar para decrementar valores de variables.

Puede ver ejemplos de esta construcción, en la definición de bucle `for` del ejemplo anterior.

El otro operador abreviado que utilizamos, es cuando agregamos un valor particular a una variable. Por ejemplo, el siguiente ejemplo:

```
precio_casa = precio_casa + precio_ventana;
```

Es perfectamente VÁLIDO, pero de nuevo da la impresión de ser una sentencia demasiado larga. C# tiene algunos operadores adicionales que permiten acortarlas:

```
precio_casa += precio_ventana;
```

El operador `+=` combina la adición y la asignación, así que el valor de `precio_casa` es incrementado con el valor de `coste_ventana`. Algunos otros operadores abreviados son:

a += b	el valor en a es reemplazado por a + b
a -= b	el valor en a es reemplazado por a – b
a /= b	el valor en a es reemplazado por a / b
a *= b	el valor en a es reemplazado por a * b

Existen otros operadores de combinación; ¡Dejare que los descubra por sí mismo!

Sentencias y Valores

En C#, todas las sentencias devuelven un valor, que puede ser utilizado en otra sentencia si así lo requiere. La mayoría de las veces ignorará este valor, lo cual es VÁLIDO, pero a veces éste puede sernos de mucha utilidad, sobre todo cuando estamos tomando decisiones (lo veremos más adelante). Con el fin de mostrar cómo se hace esto, considere lo siguiente:

```
i = (j=0);
```

Esta sentencia perfectamente válida en C# (y quizás incluso razonable), tiene el efecto de establecer tanto `i` como `j` en `0`. Una instrucción de asignación siempre devuelve el valor que se está asignando (es decir, la parte derecha que es la que realiza la operación). Este valor puede ser entonces utilizado como un valor u operando. Si hace esto, se le aconseja colocar paréntesis alrededor de la sentencia que está utilizando como valor, ¡esto hace que todo sea más claro tanto para usted como para el compilador!

Cuando se consideran operadores como **++** hay una posible ambigüedad, en la forma en que usted debe determinar si desea obtener un valor antes o después de la operación de incremento o decremento, mediante el correcto posicionamiento de los operadores abreviados. C# proporciona una manera de obtener el valor antes o después de realizar el incremento o decremento, dependiendo de lo que queramos:

`i++` Significa *"Dame el valor antes de realizar el incremento"*

`++i` Significa *"Dame el valor después de realizar el incremento"*

Como ejemplo:

```
int i = 2, j;
j = ++i;
```

- haría que `j` fuese igual a `3`. Los otros operadores especiales, **+=** etc. devuelven el valor después de que la operación se haya realizado.

Punto del Programador: Mantenga el código simple

El hecho de que pueda producir un código como este:
altura = anchura = velocidad = contador = numero = 0 ;

- no significa que deba hacerlo. Cuando estoy desarrollando un programa mi primera consideración es si el programa es fácil de entender o no. No creo que la declaración anterior sea muy fácil de seguir – por lo que independientemente de lo eficiente que sea, no veo ningún beneficio en utilizarla.

2.3.5 Imprimir por pantalla de una forma más elegante y ordenada

Si ha ejecutado cualquiera de los programas anteriores, habrá descubierto que por ahora la forma en que se imprimen los números deja mucho que desear. Los números enteros parecen tener una salida por pantalla correcta, pero los números punto flotante parecen tener un comportamiento propio. Para resolver esto, C# proporciona una forma ligeramente diferente en la que poder imprimir números por pantalla. Esta característica proporciona una mayor flexibilidad, y es mucho más útil si tiene que imprimir por pantalla un gran número de valores.

Tenga en cuenta que la forma en que se imprime un número no afecta a la forma en la que es almacenado por el programa.

Aplicar Marcadores de Posición a Cadenas de Impresión

Un marcador de posición simplemente marca el lugar donde el valor se va a imprimir. Considere:

```
int i = 150;
double f = 1234.56789;
Console.WriteLine( "i: {0} f: {1}", i, f );
Console.WriteLine( "i: {1} f: {0}", f, i );
```

El ejecutar el código anterior, obtendríamos el siguiente resultado por pantalla:

```
i: 150  f: 1234.56789
i: 150  f: 1234.56789
```

La parte `{n}` de la cadena indica el "parámetro número n, contando desde 0". En la segunda sentencia he cambiado el orden de los números, pero como también he cambiado el orden de los parámetros, la salida por pantalla es la misma.

Si realizo alguna locura, como por ejemplo utilizar `{99}` para intentar obtener el parámetro nonagésimo noveno, el método `WriteLine` fracasará en el intento y provocará un error. Este error **no** será detectado por el compilador, sin embargo, el programa fallará cuando se ejecute.

Ajustar la precisión de un número real mediante *Marcadores de Posición*

Los marcadores de posición pueden tener información de formato agregada a ellos:

```
int i = 150;
double f = 1234.56789;
Console.WriteLine( "i: {0:0} f: {1:0.00}", i, f );
```

La salida por pantalla será:

```
i: 150 f: 1234.57
```

Los caracteres `0` reemplazan el cero por el dígito correspondiente si hay alguno presente; de lo contrario, el cero aparece en la cadena de resultado. Cuando se colocan después de un punto decimal, éstos pueden ser utilizados para definir el número de posiciones decimales del valor. Tenga en cuenta que al hacer esto, si el número es un entero, será imprimido por pantalla como `12.00`.

Especificar el número de dígitos impresos

Puedo especificar un número concreto de dígitos a imprimir, estableciendo el mismo número de ceros:

```
int i = 150;
double f = 1234.56789;
Console.WriteLine( "i: {0:0000} f: {1:00000.00}", i, f );
```

La salida por pantalla será:

```
i: 0150 f: 01234.57
```

Tenga en cuenta que al hacer esto, añadimos ceros impresos a la izquierda, lo cual es muy útil si necesita imprimir cosas como cheques.

Formato Realmente Elegante

Si quiere tener un nivel elevado de control, puede utilizar el carácter #. Al incluir un carácter # se reemplaza el símbolo "#", por el dígito correspondiente si hay alguno presente; de lo contrario, no aparece ningún dígito en la cadena de resultado.:

```
int i = 150;
double f = 1234.56789;
Console.WriteLine( "i: {0:#,##0} f: {1:##,##0.00}", i, f );
```

He utilizado el carácter # para obtener unidades de miles impresas con comas:

```
i: 150 f: 1,234.57
```

Tenga en cuenta que el formateador solamente utiliza los carácteres **#** y las comas que éste necesita. El valor **150** no cuenta con ninguna unidad de miles, por lo que la coma se queda fuera.

Tenga en cuenta que también he incluido un **0** como el dígito más pequeño. Esto es así para que cuando yo imprima el valor **0**, realmente obtenga un valor impreso; de no hacerlo así cuando yo imprimiera cero, no obtendría nada en la página.

Impresión en columnas

Finalmente puede añadir un valor de ancho a la información del diseño de impresión. Esto es muy útil si desea imprimir los datos en columnas:

```
int i = 150;
double f = 1234.56789;
Console.WriteLine( "i: {0,10:0} f: {1,15:0.00}", i, f );
Console.WriteLine( "i: {0,10:0} f: {1,15:0.00}", 0, 0 );
```

Esto produciría la siguiente salida:

```
i:       150 f:       1,234.57
i:         0 f:           0.00
```

El valor entero se imprimiría en una columna de 10 caracteres de ancho, y el valor double se imprimiría en una columna de 15 caracteres de ancho. Por ahora, la salida por pantalla es justificada a la derecha. Si quiero que los números estén justificados a la izquierda, debo utilizar valores de anchos negativos:

```
int i = 150;
double f = 1234.56789;
Console.WriteLine( "i: {0,-10:0} f: {1,-15:0.00}", i, f );
Console.WriteLine( "i: {0,-10:0} f: {1,-15:0.00}", 0, 0 );
```

Esto produciría la siguiente salida:

```
i: 150        f: 1234.57
i: 0          f: 0.00
```

Tenga en cuenta que esta justificación, funcionaría incluso si estuviera imprimiendo una cadena, por lo que, si desea imprimir columnas de palabras, también puede utilizar esta técnica para hacerlo.

Puede especificar la anchura de impresión de cualquier elemento, incluso de un trozo de texto, lo que hace que imprimir en columnas sea muy sencillo de realizar.

3 Creación de programas

En este capítulo, desarrollaremos nuestras habilidades como programador dividiendo nuestros programas en partes más pequeñas, con lo que obtendremos un código más manejable y legible. También veremos cómo un programa puede almacenar y manipular grandes cantidades de datos, mediante el uso de matrices.

3.1 Métodos

Hasta ahora hemos visto y trabajado con los métodos `Main`, `WriteLine` y `ReadLine`. `Main` es el método desde el que comienza a ejecutarse nuestro programa. `WriteLine` y `ReadLine` fueron desarrollados por los creadores de C# para proporcionar una manera de mostrar y leer información de texto respectivamente.

Sus programas contendrán métodos que usted debe crear para resolver partes del problema, y éstos a su vez harán uso de otros métodos que han sido desarrollados por otras personas. En esta sección vamos a considerar por qué los métodos son necesarios, y cómo puede crear sus propios métodos.

3.1.1 La necesidad de utilizar Métodos

En el programa del acristalamiento, pasamos la mayor parte del tiempo comprobando los valores de entrada y asegurándonos de que éstos se encuentren entre ciertos rangos. Tenemos exactamente el mismo fragmento de código para comprobar anchuras y alturas. Si añadimos un tercer elemento para leerlo y comprobarlo, por ejemplo, el grosor del marco, tendríamos que copiar el código una tercera vez. Esto no es muy eficiente; ya que hace que el programa sea más grande y más difícil de escribir. Lo que nos gustaría hacer es escribir el código de comprobación una sola vez para luego usarlo en cada punto necesarios del programa. Para ello es necesario definir un método que realice este trabajo.

Los Métodos y la pereza

Ya hemos indicado que un buen programador es creativamente perezoso. Uno de los principios de esto, es que un programador va a tratar de hacer un trabajo dado de una vez, y solamente una vez.

Hasta ahora todos nuestros programas han tenido un solo método. Me refiero al bloque de código que sigue a la parte principal (`Main`) en nuestro programa. No obstante, C# nos permite crear otros métodos para utilizarlos en nuestro programa. Los métodos nos dan a nosotros dos nuevas bazas:

- Podemos usar métodos para reutilizar un trozo de código que ya hemos escrito.
- También podemos utilizar métodos para dividir una gran tarea en otras más pequeñas.

Necesitaremos contar con ambas bazas, a la hora de escribir programas más grandes. La ventaja principal de utilizar métodos, es que éstos nos ayudan a organizar mejor nuestros programas

Básicamente, un método se basa en tomar un bloque de código y otorgarle un nombre. Una vez realizado esto, puede hacer referencia a este bloque de código para que realice alguna tarea en el programa. Veamos, un ejemplo absurdo:

```
using System;

class MetodoDeDemostracion
{
	static void hazesto()
	{
		Console.WriteLine("Hola");
	}
	public static void Main()
	{
		hazesto();
		hazesto();
		Console.ReadKey();
	}

}
```

Código de Ejemplo 09 Método simple

En el método principal `Main`, hago dos llamadas al método `hazesto`. Cada vez que llamo a este método, el código que se encuentra dentro del bloque, que es el cuerpo del método, es ejecutado. En este caso el método contiene una única instrucción que imprime la cadena `"Hola"` en la consola. El resultado de ejecutar el programa anterior sería:

```
Hola
Hola
```

De esta manera, podemos utilizar los métodos para evitar tener que escribir el mismo código dos veces. Tan solamente, tendremos que escribir el código dentro del cuerpo del método, y llamar a éste cuando lo necesitemos.

3.1.2 Parámetros

En este punto, entenderá que los métodos son útiles porque nos permiten utilizar el mismo bloque de instrucciones en cualquier punto de nuestro programa. Sin embargo, los métodos se vuelven todavía muchos más útiles cuando asignamos a ellos parámetros.

Un parámetro significa pasar un valor dentro de la llamada a un método. Se puede decir, que el método recibe los datos con los que debe trabajar. Como ejemplo, considere el siguiente código:

```
using System;

class MetodoDeDemostracion
{
	static void absurdo( int i )
	{
		Console.WriteLine( "i vale: " + i );
	}

	public static void Main()
	{
		absurdo(101);
		absurdo(500);
		Console.ReadKey();
	}

}
```

Código de Ejemplo 10 Método con parámetros

El método `absurdo` tiene un único parámetro de tipo entero. Dentro del bloque de código, el cual es el cuerpo de este método, podemos utilizar el parámetro `i` como una variable de tipo entero. Cuando el método comienza, el valor suministrado para el parámetro es copiado en ésta. Por tanto, cuando el programa se ejecuta obtenemos una salida como esta:

```
i vale: 101
i vale: 500
```

3.1.3 Retornar valores

Cuando un método finaliza su trabajo, debe devolver un valor explícito al programa. Ya hemos utilizado esta característica anteriormente, los métodos `ReadLine` y `Parse` devuelven resultados que hemos utilizado en nuestros programas:

```
using System;

class DemostracionRetornarValor
{
	static int absurdoRetornoDeSuma(int i)
	{
		i = i + 1;
```

```
            Console.WriteLine ("i vale: " + i);
            return i;
        }

        public static void Main()
        {
            int resultado;
            resultado = absurdoRetornoDeSuma(5);
            Console.WriteLine("El resultado es: " + resultado);
            Console.ReadKey();
        }
    }
```

Código de Ejemplo 11 Método Retorno de suma

El método `absurdoRetornoDeSuma`, toma el valor del parámetro y devuelve el resultado tras haberle sumado uno.

El valor que un método retorna, puede ser utilizado en cualquier parte del programa donde pueda usar una variable de ese tipo. En otras palabras, la llamada al método `absurdoRetornoDeSuma` retornaría un valor de tipo entero, que podría utilizar posteriormente para realizar otra tarea.
A ver si es capaz de adivinar lo que el siguiente código devolvería como resultado:

```
resultado = absurdoRetornoDeSuma(5) + absurdoRetornoDeSuma(7) + 1;
Console.WriteLine ("El resultado es: " + resultado);
```

En realidad, se considera válido ignorar el valor devuelto por un método y no utilizarlo para nada más en su programa, pero es algo que como entenderá no tiene mucho sentido:

```
absurdoRetornoDeSuma(5); // compilará, pero no hará nada útil
```

3.1.4 Argumentos y Parámetros

Cuando lea otros libros o documentos sobre C#, descubrirá que las palabras parámetro y argumento se utilizan indistintamente como si fueran una misma cosa. Esto en realidad, no es correcto. Entender la diferencia entre ambos términos, hará que encuentre más sensata la documentación técnica de C# y los mensajes de error lanzados por el compilador.

Un **parámetro** es un tipo especial de variable que se define en el encabezado del método, y que es utilizado dentro de ese método para representar el valor incluido en la llamada.

```
static int absurdoRetornoDeSuma(int i)
{
    i = i + 1;
    Console.WriteLine("i vale: " + i);
    return i;
}
```

El anterior método `absurdoRetornoDeSuma` tiene un solo parámetro, que es de tipo entero y utiliza `i` como identificador.

Un **argumento** es el valor que se envía al método cuando se le llama.

```
absurdoRetornoDeSuma(99);
```

En la sentencia anterior, el argumento es el valor `99`. Por lo que si hago algo sin sentido como:

```
absurdoRetornoDeSuma("banjo");
```

- Obtengo el siguiente mensaje de error cuando trate de compilar el código:

```
Error 2  Argumento 1: no se puede convertir de 'string' a 'int'
```

El mensaje anterior me está indicando que el argumento (lo que puse entre los paréntesis al realizar la llamada al método), no acepta la definición del parámetro (definida como de tipo entero).

3.1.5 Un método útil

Ahora podemos empezar a escribir métodos verdaderamente útiles:

```
static double leerValor (
    string prompt,  // mensaje para el usuario desde la línea de comandos
    double minimo,  // valor mínimo permitido
    double maximo   // valor máximo permitido
    )
{
    double resultado = 0;
    do
    {
        Console.WriteLine (prompt +
           " entre " + minimo +
           " y " + maximo );
        string cadenaResultado = Console.ReadLine();
        resultado = double.Parse(cadenaResultado);
    } while ( (resultado < minimo) || (resultado > maximo) );
    return resultado;
}
```

Nota del traductor: He decidido no traducir el término `prompt` por no tener una traducción tan directa al castellano. Si lo tradujera no debería de utilizar un nombre de variable tan grande como, por ejemplo: `mensajeParaUsuario`, `mensajeParaUsuarioDesdeLíneaDeComandos` o `mensajeEsperaAccionDeUsuario`.

El método `leerValor` indica al usuario desde la ventana de comandos los valores máximo y mínimo permitidos. A continuación, empezará a leer valores introducidos por el usuario, asegurándose que se encuentren dentro del rango permitido.

```
double anchuraVentana = leerValor(
      "Introduzca la anchura de la ventana: ", MIN_ANCHURA, MAX_ANCHURA);
double edad = leerValor( "Introduzca su edad: ", 0, 70);
```

La primera llamada que realizamos al método `leerValor`, obtiene la anchura de una ventana. La segunda lee una edad entre 0 y 70.

```
using System;

class MetodoUtil
{
      static double leerValor (
        string prompt,  // mensaje para el usuario desde línea de comandos
        double minimo,  // valor mínimo permitido
        double maximo   // valor máximo permitido
        )
      {
         double resultado = 0;
         do
         {
             Console.WriteLine (prompt +
                 " entre " + minimo +
                 " y " + maximo );
             string cadenaResultado = Console.ReadLine();
             resultado = double.Parse(cadenaResultado);
         } while ( (resultado < minimo) || (resultado > maximo) );
         return resultado;
      }

      const double MAX_ANCHURA = 5.0;
      const double MIN_ANCHURA = 0.5;


      public static void Main()
      {
           double anchuraVentana = leerValor(
             "Introduzca anchura de ventana: ", MIN_ANCHURA, MAX_ANCHURA);

           Console.WriteLine("resultado es : " + res );

           double edad = leerValor("Introduza su edad: ", 0, 70);

           Console.WriteLine("Edad: " + edad);
      }

}
```

Código de Ejemplo 12 Utilizar un Metodo útil

Punto del Programador: Diseñe y Desarrolle utilizando métodos

Los métodos son una parte esencial del kit de herramientas del programador. Estos forman una parte importante del proceso de desarrollo. Una vez que ha entendido lo que el cliente desea y ha recopilado la información de los metadatos que necesita, puede empezar a pensar sobre cómo dividir el código de su programa en distintos métodos. Con frecuencia se dará cuenta al escribir el código, que está repitiendo una acción en particular varias veces. Esto es una señal clara de que ese código repetitivo, debería estar dentro de un método. Hay dos razones por lo que hacer esto así, es una buena idea:

1: Le ahorra escribir el mismo código dos veces.
2: Si encuentra un fallo en el código, solamente tiene que arreglarlo en un solo lugar.

El proceso que consiste en mejorar el código, cambiando su estructura interna sin modificar su comportamiento externo es llamado *refactorización*. La refactorización es una parte importante de la Ingeniería del Software que veremos más adelante.

3.1.6 Argumentos opcionales y con nombres identificativos

Cuando se crea un método que tiene parámetros, debemos indicar en orden secuencial al compilador, el nombre y tipo de cada parámetro:

```
static double leerValor (
  string prompt,  // mensaje para el usuario desde línea de comandos
  double minimo,  // valor mínimo permitido
  double maximo   // valor máximo permitido
  )
{
  …
}
```

El método `leerValor` ha sido definido con tres parámetros. Una llamada a este método debe tener los valores destinados a los tres argumentos en orden secuencial: uno para el prompt, otro para el valor mínimo y un tercero para el valor máximo.

Esto significa que si realizamos una llamada al método leerValor como sigue a continuación, el compilador lo rechazaría:

```
x = leerValor(25, 100, "Introduzca su edad");
```

Esto es debido a que la cadena prompt debe ser pasada en primer lugar, seguido por los límites de los valores mínimo y máximo.
Si desea realizar llamadas a métodos y no tener que preocuparse por el orden de los argumentos, puede indicar los nombres identificativos de cada uno de ellos en la llamada al método:

```
x = leerValor(minimo:25, maximo:100, prompt: "Introduzca su edad");
```

Ahora el compilador está utilizando el nombre de cada argumento, en lugar de utilizar su verdadera posición en la lista. Esto tiene el efecto secundario útil de hacer el código mucho más entendible a la hora de leer el código, ya que así sabe al instante el significado exacto de cada valor pasado como argumento.

Punto del Programador: Es preferible utilizar argumentos con nombre

Me encanta esta característica de C#. Esta hace que el programa sea mucho más claro de leer, y le ayuda a no tener que comerse la cabeza preguntándose si cuando creó el método, estableció que el valor más bajo fuese pasado antes que el valor más alto.

Argumentos opcionales

A veces el valor de un argumento puede tomar un valor por defecto razonable. Por ejemplo, si quisiéramos que el método `leerValor` recogiera un valor del usuario sin mostrar un prompt, podríamos hacerlo proporcionando una cadena vacía:

```
x = leerValor(minimo:25, maximo:100, prompt: "");
```

Sin embargo, este código es un poco confuso. En su lugar, podemos modificar la definición del método enviando un valor por defecto para el parámetro `prompt`:

```
static double leerValor (
  double minimo,  // valor mínimo permitido
  double maximo   // valor máximo permitido
  string prompt = "",  // mensaje opcional para el usuario
  )
{
  …
}
```

Ahora podemos invocar al método, sin incluir el parámetro `prompt`, si así lo deseamos:

```
x = leerValor(25, 100);
```

Cuando el método ejecute el mensaje desde la línea de comandos, se establecerá a una cadena vacía si el usuario no proporciona un valor.
Tenga en cuenta que he tenido que reordenar el orden de los parámetros, para que el mensaje sea el último parámetro que envíemos en la llamada al método. Los parámetros opcionales siempre deben ser proporcionados después de los requeridos.
Sin embargo, aquí nos encontramos con un problema potencial. Considere este método:

```
static double leerValor (
  double minimo,  // valor mínimo permitido
  double maximo   // valor máximo permitido
  string prompt = "",  // mensaje opcional para el usuario
  string error = ""// mensaje de error opcional
  )
{
  …
}
```

Este tiene dos parámetros opcionales, un mensaje para indicarle al usuario que estamos a la espera de órdenes y un mensaje de error. La idea es que el método pueda proporcionar al usuario un mensaje de error personalizado, si el usuario introduce una edad no válida. Ahora podemos llamar al método de la siguiente manera:

```
x = leerValor(25, 100, " Introduzca su edad", "Edad fuera de rango");
```

Si no considero necesario establecer un mensaje para indicarle al usuario que estamos a la espera de órdenes ni proporcionar un mensaje de error, puedo omitirlo. Pero si solamente envío un argumento, éste se asigna al primero en la secuencia de parámetros opcionales. En otras palabras, no puedo proporcionar un mensaje de error personalizado, sin proporcionar un prompt personalizado.

La manera de solucionarlo (como estoy seguro que habrá deducido), es identificar los parámetros opcionales que desea utilizar mediante un nombre que los identifique.

```
x = leerValor(25, 100, error:" Edad fuera de rango");
```

Esta llamada al método `leerValor`, utilizaría el prompt por defecto, pero contaría con un mensaje de error personalizado.

Punto del Programador: No soy muy partidario de utilizar valores por defecto en los parámetros

Por ahora espero que esté pensando que la programación es tan artesanal como cualquier otra cosa. Esto significa que debemos analizar ciertos comportamientos y considerarlos dentro del contexto de la destreza de escribir código, de la misma manera que le podría preocupar si alguien le pidiera que le hiciera una pieza de ajedrez, utilizando una motosierra para ello.

Los valores por defecto de los parámetros utilizados en los métodos, pueden ocultar información a los usuarios. Estos pueden proporcionar "interruptores secretos" que hacen que las cosas funcionen de una manera particular. Le aconsejo añadir comentarios tanto en el método que los define como en el código que los utiliza, para que alguien que esté interesado en leer su código comprenda lo que hacen los comportamientos predeterminados y cómo éstos pueden ser modificados.

3.1.7 Paso de parámetros por valor

Un método es útil y efectivo a la hora de ejecutar un trabajo que tengamos que realizar varias veces, pero puede ser un poco limitante debido a la manera en que trabaja. Por ejemplo, si necesito escribir un método que lea el nombre y la edad de una persona, tengo un problema, dado que como hemos visto anteriormente los métodos, solamente pueden devolver un valor. Así que tengo que crear un método que retorne el nombre de una persona, y otro método que devuelva la edad. Pero no puedo escribir un método que devuelve ambos valores al mismo tiempo, ya que un método solamente puede devolver un valor. Como comprobará cuando lo pruebe, cambiar el valor de un parámetro no cambia el valor del elemento enviado al método. Esto se debe a que, a menos que especifique lo contrario, solamente el **valor** de un argumento es pasado cuando se realiza la llamada a un método.

Entonces, a qué me refiero cuando digo "pasar parámetros por valor". Considere el siguiente código:

```
static void anadirUno(int i)
{
    i = i + 1;
    Console.WriteLine("i vale: " + i );
}
```

El método `anadirUno`, añade uno al parámetro; posteriormente imprime el resultado por pantalla y finalmente retorna:

```
int prueba = 20 ;
anadirUno(prueba);
Console.WriteLine("prueba vale: " + prueba);
```

El código anterior, llama al método utilizando la variable `prueba` como argumento. Cuando éste retorna, imprime lo siguiente:

```
i vale: 21
prueba vale: 20
```

Es **muy importante** que entienda lo que está ocurriendo aquí. El **valor** de `prueba` está siendo utilizado en la llamada al método `anadirUno`. El programa calcula el resultado de la expresión que se pasa a la llamada del método como argumento. A continuación, pasa este valor en la llamada. Esto significa que si escribe un código de llamada como este:

```
prueba = 20;
anadirUno(prueba + 99);
```

Se imprimirá como resultado:

```
i vale: 120
```

El paso por valor es muy seguro, debido a que nada de lo que se haga en el método, puede afectar a las variables pertenecientes al código desde el que se llama. Sin embargo, esto es una limitación cuando queremos crear un método que devuelve más de un valor.

3.1.8 Paso de parámetros por referencia

Afortunadamente C# proporciona una manera para que, en lugar de enviar el valor de una variable al método, se envíe una referencia de la misma. Dentro del método, en lugar de utilizar el valor de la variable, se proporciona una referencia directa al elemento. Técnicamente lo que se pasa al método es la posición o dirección que ocupa la variable en la memoria.

Así que, en nuestro caso anterior, en lugar de pasar "20" a la llamada, el compilador generará un código que enviará el mensaje "Dirección de memoria 5023" (suponiendo que la variable `prueba` se encuentre almacenada en esa dirección 5023). Esta dirección de memoria es utilizada por el método, en lugar del valor. En otras palabras:

"Si pasa por referencia, los cambios en el parámetro cambian la variable cuya referencia haya pasado"

Si encuentra los pasos por referencia algo confusos, puede estar tranquilo porque no es la primera persona a la que le ocurre. Sin embargo, va a ver que nosotros utilizamos las referencias en nuestra vida real con bastante frecuencia. Si usted dice "Entregar la alfombra en el número 23 de High Street.", le está dando al repartidor una referencia. El uso de una referencia en un programa es exactamente la misma cosa. El programa va a decir al compilador "Obtener el valor que hay en la dirección de memoria 5023", en lugar de decirle "El valor es 1".

Considere el siguiente código:

```
static void anadirUnoAlParamRef( ref int i )
{
      i = i + 1;
      Console.WriteLine("i vale: " + i);
}
```

Tenga en cuenta que la palabra reservada `ref`, ha sido añadida a los datos del parámetro.

```
prueba = 20 ;
anadirUnoAlParamRef(ref prueba);
Console.WriteLine("prueba vale: " + prueba);
```

El código anterior realiza una llamada al nuevo método, que como observará incorpora la palabra reservada `ref` delante del parámetro. En este ejemplo, la salida por pantalla será la siguiente:

```
i vale : 21
prueba vale : 21
```

En este caso la llamada al método ha realizado cambios en el contenido de la variable. Tenga en cuenta que C#, presta especial atención cuando un parámetro es una referencia. Hay que escribir el modificador `ref` tanto en el encabezado del método como en la llamada al método.

Punto del Programador: Documente los efectos colaterales

Una modificación de un método al estado de su entorno, es denominada un ***efecto colateral*** del método. En otras palabras, cuando se realiza una llamada a nuestro método `anadirUnoAlParamRef`, éste va a provocar el "efecto secundario" de cambiar algo fuera del propio método en sí (concretamente el valor del parámetro pasado por referencia). Generalmente, hay que tener cuidado con los efectos colaterales, por lo que hay que informar mediante comentarios, que el método realiza cambios de este tipo.

3.1.9 Pasar parámetros por referencia, utilizando el modificador "out"

Cuando envía un parámetro como referencia, está dando al método un control completo sobre éste. En ciertas ocasiones, queremos que esto no sea así. Este es el caso, por ejemplo, cuando queremos leer el nombre y la edad de un usuario, donde tan solamente estamos interesados en que se nos devuelva los resultados introducidos por el usuario. En este caso puede reemplazar el modificador `ref` por el modificador `out`.

El modificador **out** puede ser útil para retornar múltiples valores desde un método. Ejemplo:

```
static void leerPersona( out string nombre, out int edad )
{
      nombre = leerCadena ( "Introduzca su nombre : " );
      edad = leerEntero( "Introduzca su edad : ", 0, 100 );
}
```

El método `leerPersona` lee el nombre y la edad de una persona. Tenga en cuenta que éste, utiliza dos métodos más que yo he creado, `leerCadena` y `leerEntero`.
Puedo invocar al método `leerPersona` de la siguiente manera:

```
string nombre;
int edad;
leerPersona( out nombre, out edad );
```

Como comprobará, también debo utilizar la palabra reservada `out` en la llamada al método.

El método `leerPersona` leerá los datos de una persona y entregará la información en las dos variables correspondientes.

Punto del Programador: Los lenguajes de programación pueden echar un capote a los programadores

La palabra reservada `out` es un buen ejemplo de cómo el diseño de un lenguaje de programación puede hacer que los programas sean más seguros y fáciles de escribir. Este modificador se asegura de que un programador no pueda cambiar el valor del parámetro en el método. También permite al compilador, asegurarse que en alguna parte del método se asignen valores a los parámetros de salida. Esto es muy útil. Significa que, si yo marco los parámetros como `out`, **debo** darles un valor para que el programa compile. Esto dificulta que el programa pueda recibir un valor erróneo, ya que estaré protegiéndome de olvidar de hacer esa parte del trabajo.

3.1.10 Bibliotecas de Métodos

La primera cosa que un buen programador hará cuando comience a escribir código es crear un conjunto de bibliotecas que puedan ser utilizadas para hacer más fácil su trabajo. En el código anterior, he escrito un par de métodos de biblioteca, que puedo utilizar para leer valores de diferentes tipos:

```
static string leerCadena(string prompt)
{
      string resultado;

      do
      {
           Console.Write(prompt) ;
           resultado = Console.ReadLine();
      } while (resultado == "") ;

      return resultado ;
}

static int leerEntero(string prompt, int minimo, int maximo)
{
      int resultado;

      do
      {
           string cadenaEntero = leerCadena (prompt);
           resultado = int.Parse(cadenaEntero);
      } while ( (resultado < minimo ) || (resultado > maximo) );

      return resultado;
}
```

El método leerCadena leerá el texto y se asegurará de que el usuario no haya introducido una cadena vacía. El método `leerEntero`, leerá un número dentro de un rango determinado. Observe cómo he incluido ingeniosamente el método `leerCadena` dentro del método `leerEntero`, de modo que el usuario no puede introducir una cadena vacía cuando realmente un número es requerido. Podemos utilizar los métodos de la siguiente manera:

```
string nombre;
nombre = leerCadena("Introduzca su nombre: ");

int edad;
edad = leerEntero("Introduzca su edad: ", 0, 100);
```

También podría añadir métodos para leer valores de punto flotante. De hecho, una cosa que tiendo a hacer cuando trabajo en un proyecto, es crear una pequeña biblioteca de métodos tan útiles como estos, que vaya a utilizar.

```
using System;
class BibliotecaMetodos
{
      static string leerCadena(string prompt)
      {
         string resultado;

         do
         {
             Console.Write(prompt);
             resultado = Console.ReadLine();
         } while (resultado == "");

      return resultado;
      }

      static int leerEntero(string prompt, int minimo, int maximo)
      {
         int resultado;

         do
         {
             string cadenaEntero = leerEntero (prompt) ;
             resultado = int.Parse(cadenaEntero);
         } while ( ( resultado < minimo ) || ( resultado > maximo ) );

      return resultado;
      }

      public static void Main()
      {
          string nombre;
          nombre = leerCadena("Introduzca su nombre: ");
          Console.WriteLine("Nombre: " + nombre);

          int edad;
          edad = leerEntero("Introduzca su edad: ", 0, 100);
          Console.WriteLine("Edad: " + edad);
          Console.ReadKey();
      }
}
```

Código de Ejemplo 13 Biblioteca de Métodos

Punto del Programador: Siempre considere los comportamientos fallidos

Siempre que escriba un método, debe pensar de qué forma podría fallar. Si el método implica comunicarnos con el usuario, es posible que el usuario quiera abandonar el método, o que el usuario realice alguna acción que haga que falle. Siempre tiene que considerar, si el método debe o no tratar el problema por sí mismo, o pasar el error al sistema para que éste se encargue de tratarlo y actúe en consecuencia.

Si el método lidia con el error por sí mismo, puede acarrear problemas, ya que el usuario puede que no tenga ninguna forma de cancelar una acción. Si el método pasa el error al código desde el que fue llamado, debe tener un método mediante el cual se pueda enviar una condición de error al llamador. A menudo resuelvo este problema, haciendo que mis métodos devuelvan un valor. Si el valor devuelto es `0`, significa que el método ha retornado correctamente. Si el valor devuelto no es cero, significa que el método no funciona correctamente, y el valor del error que devuelve identifica lo que salió mal. Esto añade otra dimensión al diseño de los programas, en la que usted también deberá considerar cómo el código que escribe puede fallar por cualquier circunstancia, ¡así como también deberá asegurarse que éste realiza el trabajo requerido! Veremos cómo gestionar los errores más adelante.

3.2 Alcance de Variables

Hemos visto que cuando queremos guardar una cantidad en nuestro programa, podemos crear una variable para almacenar esa información. El compilador de C#, se asegura de reservar el correspondiente tamaño de memoria para almacenar el valor, y de que utilicemos ese valor correctamente. Además, el compilador de C#, también se ocupa de conocer la parte del programa donde una variable tiene vida y puede ser utilizada. A esto se le conoce como el *alcance* de una variable.

3.2.1 Ámbito de Variables en Bloques de Código

Hemos visto que un bloque de código es un grupo de instrucciones encerradas entre corchetes. Un bloque puede contener un número indeterminado de variables *locales*, es decir, variables que se encuentran dentro de ese bloque.

El *ámbito* de una variable local, es el bloque dentro del cual se declara la variable. En lo que concierne al lenguaje C#, puede declarar una variable en cualquier punto dentro del bloque, pero

debe declararla antes de poder utilizarla. Cada vez que la ejecución del programa sale de un bloque, cualquier variable local que fuera declarada en el bloque, es automáticamente desechada.
Los métodos que hasta ahora hemos creado han solido contener variables locales; la variable `resultado` dentro del método `leerEntero`, es local al bloque del método.

3.2.2 Bloques Anidados

Hemos visto que en C# los programadores pueden crear bloques de código dentro de otros bloques de código. Cada uno de estos bloques *anidados* pueden tener su propio conjunto de variables locales:

```
{
      int i;
      {
            int j;
      }
      j = 99;
}
```

`-` podría causar un error, dado que la variable `j` no existe en ese punto del programa.
Con el fin de evitar que se confunda creando dos versiones de una variable con el mismo nombre, C# incorpora una regla adicional sobre las variables internas en un bloque:

```
{
      int i;
      {
            int i;
      }
}
```

Este no es un programa válido, porque C# no permite que una variable en un bloque interno, tenga el mismo nombre que otra variable en un bloque externo. Esto se debe a que, dentro del bloque interior, existe la posibilidad de que pueda usar la versión “interna” de `i`, cuando realmente quiere usar la versión “exterior”. Para acabar con esta posibilidad, el compilador le prohíbe hacer esto. En otros lenguajes de programación, como C++, se admite la posibilidad de realizar esta acción.

En C#, sin embargo, está permitido reutilizar un nombre de variable en bloques sucesivos, ya que en esta situación no existe ninguna posibilidad de que una variable puede ser confundida con otra.

```
{
      int i;
}
{
      int i;
      {
            int j;
      }
}
```

La primera encarnación de `i` ha sido destruida antes de la segunda, por lo que este código es correcto.

Variables locales en bucles For

Un tipo especial de variable puede ser utilizada cuando creas un bucle `for`. Esto le permite declarar una variable de control que existe durante la duración del propio bucle:

```
for (int i = 0; i < 10; i = i + 1);
{
       Console.WriteLine("Hola");
}
```

La variable `i` es declarada e inicializada al inicio del bucle `for`, y solamente existe durante la ejecución del propio bloque.

3.2.3 Miembro de datos en clases

Las variables locales son muy útiles, pero son destruidas cuando la ejecución del programa, abandona el bloque de código donde han sido declaradas. Nosotros necesitaremos frecuentemente variables que existan fuera de los métodos de una clase.

Variables Locales en un Método

Considere el siguiente código:

```
class EjemploVariableLocal
{
      static void OtroMetodo()
      {
            local = 99; // esto no compilará
      }

      static void Main()
      {
            int local = 0;
            Console.WriteLine("local vale: " + local);
      }
}
```

Código de Ejemplo 14 Variable Local

La variable `local` es declarada y utilizada dentro del método `Main`, y no puede ser utilizada en otra parte. Si una instrucción en `OtroMetodo` trata de utilizar `local`, el programa no compilará.

Variables que son Miembros de Datos de una Clase

Si quiero permitir que dos métodos en una clase compartan una variable, tengo que hacer a la variable como miembro de la clase. Esto significa que tendré que declararla fuera de los métodos de la clase:

```
class EjemploMiembro
{
      // la variable miembro es parte de la clase
      static int miembro = 0 ;

      static void OtroMetodo()
      {
            miembro = 99;
      }
```

```
        static void Main()
        {
              Console.WriteLine("miembro vale: " + miembro);
              OtroMetodo();
              Console.WriteLine("miembro vale ahora: " + miembro);
              Console.ReadKey();
        }
  }
```

Código de Ejemplo 15 Miembro de Datos

La variable `miembro` es ahora parte de la clase `EjemploMiembro`, por lo que tanto los métodos `Main` y `OtroMetodo` pueden utilizar esta variable. El programa anterior imprimiría el siguiente resultado por pantalla:

```
miembro vale: 0
miembro vale ahora: 99
```

Esto se debe a que al realizar la llamada a `OtroMetodo`, se cambiaría el valor de `miembro` a `99` cuando se ejecutara.

Las variables que son miembros de datos de una clase, son muy útiles cuando se necesita tener una serie de métodos "compartiendo" un conjunto de datos. Por ejemplo, si está creando un programa para jugar al ajedrez, sería razonable que la variable que almacena el tablero sea un miembro de la clase. De esta forma, los métodos encargados de recibir los movimientos que el usuario quiere realizar, mostrar el tablero, y analizar los movimientos que tiene que hacer la computadora, podrían utilizar la misma información.

Clases y Miembros estáticos

Tenga en cuenta que he hecho el miembro de datos de la clase estático, así que éste es parte de la clase, y no una instancia de la clase. Esto no es algo por lo que deba preocuparse en estos momentos; estudiaremos el significado preciso de `static` más adelante. Por ahora, solamente tiene que recordar incluir la palabra reservada `static`, de lo contrario su programa no compilará.

Un error en programación muy común es confundir el modificador `static` con el modificador `const`. Declarar una variable como `const` significa que "el valor no puede ser cambiado". Declarar una variable como `static` significa que "la variable es parte de la clase y siempre está presente". Si le sirve de ayuda, puede imaginar `static`, como algo que siempre está con nosotros, como el ruido estático presente en el fondo de una emisora de radio cuando la sintoniza. Alternativamente, puede pensar que es estacionaria y, por lo tanto, no va a ninguna parte.

Punto del Programador: Planifique bien, el uso de las variables

Debe planificar el uso de las variables en sus programas. Deberá decidir qué variables sólo se requieren para su uso en bloques locales, y cuáles de ellas deberán ser miembros de la clase. Tenga en cuenta que, si hace que una variable sea un miembro de la clase, ésta puede ser utilizada por cualquier método existente en esa clase (lo que aumenta las posibilidades de que ocurra algo imprevisto). Personalmente, intento declarar el menor número posible de variables miembro en una clase, y en lugar de éstas, utilizar variables locales en los casos en los que necesito guardar el valor para ser utilizado por una pequeña parte del código.

3.3 Matrices

Ahora sabemos crear programas que pueden leer valores de entrada, realizar operaciones aritméticas e imprimir los resultados. Nuestros programas también pueden tomar decisiones basadas en los valores suministrados por el usuario y también repetir acciones un número determinado de veces.
Prácticamente ya conoce casi todas las características del lenguaje requeridas para poder desarrollar cualquier programa existente. Tan sólo falta una cosa, que es la capacidad de crear programas que almacenen grandes cantidades de datos. Las matrices son una manera de hacer esto, y nosotros vamos a saber más sobre ellas en la siguiente sección.

3.3.1 ¿Para qué necesitamos las Matrices?

Su fama como un programador está empezando a extenderse. La siguiente persona en visitarle y solicitar sus servicios es la persona a cargo del equipo local de cricket. Él necesitaría que le desarrollara un programa que le ayudara a analizar el rendimiento de sus jugadores. Cuando se juega un partido de cricket cada miembro del equipo anotará un número determinado de carreras. Lo que el cliente quiere es bastante simple; dado un conjunto de puntuaciones conseguidas por los jugadores en un partido, listar esas puntuaciones en orden ascendente.

Lo primero que debería hacer es refinar la especificación y añadir algunos metadatos. Discutirá con el cliente sobre los rangos razonables a ser manejados por el programa (ningún jugador puede anotar menos de `0`, o realizar más de `1000` carreras en un partido). Planificará una versión aproximada de lo que el programa aceptará, y qué información mostrará el programa en pantalla. Decidirá cuánto dinero va a cobrar por realizar el trabajo, y cuándo lo recibirá. Finalmente, para hacer de este el proyecto perfecto, negociará cuando el programa deberá estar finalizado y cuándo se lo mostrará a su cliente. Todas estas condiciones deberán ser indicadas por escrito, mediante un documento acreditativo que deberá firmar usted y su cliente. Con todos estos puntos en orden, todo lo que tendrá que hacer a partir de entonces, es escribir el programa en sí. Usted pensará, “Esto es

fácil". Una vez se encuentre desarrollando el programa, debe definir la forma en que los datos van a ser almacenados:

```
int puntuacionJugador1, puntuacionJugador2, puntuacionJugador3,
    puntuacionJugador4, puntuacionJugador5, puntuacionJugador6,
    puntuacionJugador7, puntuacionJugador8, puntuacionJugador9,
    puntuacionJugador10, puntuacionJugador11;
```

Ahora puede empezar a introducir los datos en cada variable. Puede utilizar su nuevo método de lectura de números para esto:

```
puntuacionJugador1 = leerEntero("Puntuación de Jugador 1: ", 0,1000);
puntuacionJugador2 = leerEntero("Puntuación de Jugador 2: ", 0,1000);
puntuacionJugador3 = leerEntero("Puntuación de Jugador 3: ", 0,1000);
puntuacionJugador4 = leerEntero("Puntuación de Jugador 4: ", 0,1000);
puntuacionJugador5 = leerEntero("Puntuación de Jugador 5: ", 0,1000);
puntuacionJugador6 = leerEntero("Puntuación de Jugador 6: ", 0,1000);
puntuacionJugador7 = leerEntero("Puntuación de Jugador 7: ", 0,1000);
puntuacionJugador8 = leerEntero("Puntuación de Jugador 8: ", 0,1000);
puntuacionJugador9 = leerEntero("Puntuación de Jugador 9: ", 0,1000);
puntuacionJugador10 = leerEntero("Puntuación de Jugador 10: ", 0,1000);
puntuacionJugador11 = leerEntero("Puntuación de Jugador 11: ", 0,1000);
```

Todo lo que tenemos que hacer a continuación es ordenar los resultados…. Hmmmm…. ¡Esto es terrible!, no parece haber ninguna manera de hacerlo. Tan solo para comprobar si la puntuación anotada por el primer jugador ha sido la más alta, ¡tendríamos que crear una estructura de control `if` con 10 comparaciones! Está claro que tiene que haber una forma mejor de hacer esto, después de todo, sabemos que los ordenadores son muy buenos resolviendo este tipo de tareas.

C# nos proporciona un elemento llamado *matriz*. Una matriz nos permite declarar una hilera de cajas de un tipo en particular. Una vez creada, podemos utilizar *subíndices*, para indicar qué caja de la hilera queremos utilizar. Considere el siguiente código:

```
using System;

class DemoMatriz
{
      public static void Main()
      {
          int[] puntuaciones = new int[11];
          for ( int i=0; i<11; i=i+1 )
          {
              puntuaciones[i] = leerEntero("Puntuación: ", 0,1000);
          }
      }
}
```

La parte del código `int[] puntuaciones`, indica al compilador que queremos crear una variable de matriz. Puede imaginársela, como una etiqueta que puede crear para referirse a una determinada matriz.

La parte del código que crea la matriz en sí, es la sentencia `new int[11]`. Cuando C# ve esta instrucción se dice "¡Ajá! Lo que necesitamos aquí es una matriz". Éste entonces coge algunos trozos de madera y construye una caja delgada y larga con 11 compartimientos, cada uno lo suficientemente grande como para poder almacenar un valor de tipo entero. A continuación, pinta toda la caja de color rojo - porque las cajas que pueden contener números enteros son rojas. A continuación, coge un trozo de cuerda y ata la etiqueta `puntuaciones` a esta caja. Si sigue la cuerda desde la etiqueta puntuaciones, llegará hasta la caja de la matriz. En realidad, en este proceso, no se utiliza ni madera ni cuerda, pero puede hacerse una idea de lo que está ocurriendo.

3.3.2 Elementos de una matriz

A cada uno de los compartimentos de la caja se le llama elemento. En el programa puede identificar el elemento al que quiere referirse escribiendo su número de compartimento en el interior de los corchetes [ ] después del nombre identificativo de la matriz. Esta es la parte denominada *subíndice*. Lo que realmente hace que las matrices sean tan maravillosas, es el hecho de poder especificar un elemento de la matriz, mediante el uso de una variable. De hecho, puede usar cualquier expresión que devuelva un resultado de tipo entero como subíndice, es decir:

```
puntuaciones[i+1]
```

- esto es CORRECTO (siempre y cuando no haya alcanzado el final de la matriz). Cuando una matriz es creada, todos los elementos de la matriz son establecidos a `0`.

Numeración de los Elementos de una Matriz

C# numera las cajas desde `0`. Esto significa que otorga al primer elemento de la matriz el subíndice `0`. Consecuentemente, no existe ningún elemento `puntuaciones[11]`. Si revisa la parte del programa encargada de leer los valores de la matriz, comprobará que solamente contamos del 0 a 10. Esto es muy importante. Un intento de acceso a elementos fuera de los límites de la matriz `puntuaciones`, provoca que su programa falle mientras se ejecuta. Si encuentra confuso; "El primer elemento tiene el subíndice 0", la mejor manera es que considere el subíndice, como la distancia que tiene que recorrer para obtener el elemento que necesita.

Una cosa que se añade a esta confusión, es el hecho de que este sistema de numeración no es igual en otros lenguajes de programación. Visual Basic numera los elementos de una matriz a partir de `1`. Con esto se demuestra que no todo acerca de la programación es congruente.

3.3.3 Matrices de gran tamaño

El verdadero poder de las matrices proviene de poder utilizar una variable para especificar el elemento requerido. Mediante el recorrido de la variable a través de un rango de valores, podemos explorar los valores de la matriz utilizando un código pequeño y sencillo; de hecho, para modificar el programa y que lea 1000 puntuaciones, solamente tenemos que hacer un par de cambios:

```
using System;

class DemoMatriz
{
      public static void Main()
      {
          int[] puntuaciones = new int[11];
          for ( int i=0; i<1000; i=i+1 )
          {
              puntuaciones[i] = leerEntero("Puntuación: ", 0,1000);
          }
      }
}
```

Este código establece el rango de la variable `i` de 0 a 999, incrementando la cantidad de datos que estamos almacenando.

Gestionar el Tamaño de las Matrices

Un buen truco cuando se trabaja con matrices, es hacer uso de variables constantes para crear el tamaño de la matriz. Esto tiene dos ventajas significativas:

- Hace que el programa sea más fácil de entender.
- Hace que el programa sea más fácil de modificar.

Una variable constante debe contener un valor cuando se declara. Este valor solamente puede ser leído por el programa, nunca actualizado. Así pues, si quisiera mejorar mi programa de puntuaciones para que pudiera fácilmente ser modificado para cualquier tamaño de equipo, podría hacerlo de la siguiente manera:

```
using System;

class DemoMatriz
{
      public static void Main()
      {
          const int TAMAÑO_MATRIZ_PUNTUACIONES = 1000;
          int[] puntuaciones = new int[TAMAÑO_MATRIZ_PUNTUACIONES];
          for ( int i=0; i < TAMAÑO_MATRIZ_PUNTUACIONES; i=i+1 )
          {
              puntuaciones[i] = leerEntero("Puntuación : ", 0,1000);
          }
      }
}
```

Código de Ejemplo 16 Utilizar una Matriz

La variable TAMAÑO_MATRIZ_PUNTUACIONES, es de tipo entero y ha sido definida como `const`. Esto significa que no puede ser modificada por ninguna instrucción del programa. Por lo tanto, ésta nunca tendrá un valor distinto de 1000. Hay una convención de que los nombres identificativos de las constantes, deben ser escritos en MAYÚSCULAS y con un guion bajo que separe cada una de las palabras por las que esté formada.

Aunque previamente he prefijado un valor para representar el tamaño de la matriz, en este caso he utilizado una variable constante para ello. Esto significa que, si el tamaño del equipo cambia, solamente tendré que cambiar el valor asignado a ésta en la línea en donde la constante es declarada en el código, y volver a compilar el programa. El otro beneficio de esto, es que ahora el bucle `for` es ahora un poco más coherente y significativo. Dado que el valor de `i` va ahora desde 0 a TAMAÑO_MATRIZ_PUNTUACIONES, es más evidente para el lector que está trabajando a través de la matriz de las puntuaciones.

3.3.4 Creación de una Matriz de dos dimensiones

Sin embargo, a veces queremos almacenar más de una hilera de valores, por lo que tendremos que pasar a utilizar una especie de matriz con forma de cuadrícula o rejilla. Para poder hacer esto, tendremos que crear una matriz de *dos dimensiones*. Si quiere, puede pensar en esto como una "matriz de matrices" (pero sólo si esto no hace que le duela la cabeza). Por ejemplo, para almacenar el tablero de un juego de tres en raya, podríamos utilizar las siguientes sentencias:

```
int[,] tablero = new int[3,3];
tablero[1,1] = 1;
```

Esto se parece mucho al código de nuestra matriz unidimensional, pero existen algunas diferencias importantes. La parte del código `[,]` ahora presenta una coma. La presencia de una coma implica

algo en cada lado. En este caso significa que la matriz tiene ahora dos dimensiones, en lugar de solamente una. Así que cuando establezcamos el tamaño del tablero, debemos proporcionar dos dimensiones en vez de una. Del mismo modo, cuando queramos hacer referencia a un elemento, tendremos que hacerlo a través de dos valores subíndices. En el código anterior, he establecido el valor de la casilla central de la matriz en `1`.

Puede imaginar una matriz de dos dimensiones como una cuadrícula o rejilla:

	0	1	2
0	0	0	0
1	0	1	0
2	0	0	0

El primer subíndice debe ser utilizado para indicar la fila, y el segundo subíndice para indicar la columna. El diagrama anterior muestra el tablero, tras la ejecución del código.

En el ejemplo anterior, la matriz es cuadrada (es decir, tiene la misma dimensión tanto de largo como de ancho). Pero estas dimensiones pueden ser distintas entre ellas, si así lo necesitamos:

```
int[,] tablero = new int[3,10];
```

- aunque resultaría bastante difícil jugar a un juego con cierta lógica utilizando este tablero.

3.3.5 Matrices como grandes Tablas de Consulta o Búsqueda

Las matrices son grandes tablas de consulta o búsqueda. Éstas proporcionan una manera ordenada y sencilla de utilizar un número para acceder a otro dato relacionado. Por ejemplo, es posible que desee imprimir el nombre de un mes, utilizando para ello el valor numérico del mes. Podría intentarlo, por ejemplo, de la siguiente forma:

```
int numeroDelMes = 1;
string nombreDelMes;

if (numeroDelMes == 1)
    nombreDelMes = "Enero";

if (numeroDelMes == 2)
    nombreDelMes = " Febrero";
```

Sin embargo, este método sería tedioso de escribir. Una manera más organizada y sencilla sería utilizar una matriz a la que podría denominar `nombresMeses`. Podríamos hacer algo como esto:

```
nombreDelMes = nombresMeses[numeroDelMes];
```

Para que esto funcione deberíamos asegurarnos de que cada uno de los elementos pertenecientes a la matriz `nombresMeses`, almacenan el nombre de los meses en la matriz.

```
string[] nombresMeses = new string[13];
```

Así pues, antes de poder utilizar la matriz como tabla de consulta, tendríamos que establecer los valores correspondientes en la matriz.

```
nombresMeses[1] = "Enero";
nombresMeses[2] = "Febrero";
```

Punto del Programador: A veces es mejor "desperdiciar" el elemento 0 de la matriz

Si ha seguido cuidadosamente el texto hasta ahora, habrá detectado algo extraño en el código anterior. Como he indicado en anteriores ocasiones, las matrices son indexadas a partir de `0`, en otras palabras, el primer elemento de la matriz `nombresMeses` tendrá el subíndice `cero`, lo que significa que realmente debería de haber almacenado la cadena "`Enero`" dentro de `nombresMeses[0]`. Esto es cierto si quiero hacer un uso más eficiente del almacenamiento en memoria, pero también hace que mi programa sea más difícil de entender, dado que uso la matriz para decodificar el valor de un mes (que por supuesto será de `1` a `12`). Por lo tanto, tendría que restar 1 del valor. Para este caso, considero razonable crear una matriz que sea más grande que la requerida (de modo que los subíndices vayan de 0 a 13), e ignorar el primer elemento de la matriz.

Un programador purista, podría señalar la cuestión de que si el programa intenta utilizar el elemento base en la matriz de nombres (el que tiene el subíndice `0`), corre el peligro de mostrar un valor de mes inválido. Podemos protegernos de esto, haciendo que el elemento apunte a una referencia `null`.

```
nombresMeses[0] = null;
```

Esto significa que, si el código intenta utilizar este elemento de la matriz, lanzará una excepción y se detendrá.

Si el programador purista continúa quejándose de que hacer esto así podría provocar que los programas se comporten de manera errónea, podemos pedirle que considere el efecto de un programador que omita la parte "mes menos uno", cuando se decodifica el valor de un mes. En este caso, obtendríamos impreso el nombre del mes equivocado, y el programa continuaría ejecutándose. Prefiero tener un programa que explote formando una lluvia de llamas y chispas, que otro que haga algo incorrecto de manera silenciosa.

En este momento, resulta obvio que hemos intercambiado un fragmento de código tedioso (realizando montones de pruebas para obtener la cadena del mes) por otro (inicializando muchos elementos de la matriz). Sin embargo, resulta que C# tiene una manera efectiva y ordenada de establecer los elementos en una matriz.

```
string[] nombresMeses = new string[];
{
    null, // elemento nulo para el mes inexistente 0
    "Enero", "Febrero", "Marzo", "Abril",
    "Mayo", "Junio", "Julio", "Agosto",
    "Septiembre", "Octubre", "Noviembre", "Diciembre"
}
```

Un programa puede contener una lista de valores de inicialización, que posteriormente pueden ser utilizados para crear una matriz con el contenido requerido. Tenga en cuenta que el proceso de inicialización de la matriz calculará automáticamente incluso la longitud que ésta debe tener para almacenar los valores, si fuese necesario. También puede inicializar una matriz 2D de la siguiente forma:

```
int[,] valoresDePonderacion = new int[3,3]
{
    {1,0,1},
    {0,2,0},
    {1,0,1}
}
```

El código anterior crea una matriz 2D, que contiene valores de ponderación para realizar un programa inteligente que permite jugar al tres en raya. Éste representa mi creencia de que las casillas del centro y las esquinas tienen mayor peso en el juego, y por tanto son las más importantes. Cuando mi programa tenga que realizar su primer movimiento, tratará de posicionar la ficha en la casilla libre del tablero que tenga el valor más alto.

3.3.6 Más allá de las dos dimensiones

Esporádicamente es posible que necesite utilizar más de dos dimensiones. Si recurre a las tres dimensiones puede imaginarlo como una serie de rejillas bidimensionales colocadas una sobre otra, proyectando la tercera dimensión (que puede llamar z), la rejilla que necesita. Si quisiéramos jugar al tres en raya en un tablero tridimensional, el cual tendría forma de cubo, podríamos declarar una matriz adecuada para el tablero, de la siguiente manera:

```
int[,,] tablero = new int[3,3,3];
tablero [1,1,1] = 1;
```

Este código crea un tablero tridimensional, y a continuación, obtiene la ubicación que tiene el valor más alto en el medio del cubo.

Podemos utilizar dimensiones superiores a tres, si lo necesitamos. Sin embargo, es posible que esto pueda ocasionarle grandes problemas, ya que suele ser difícil de entender y visualizar.

Debe ser consciente de que a veces podría pensar que necesita añadir una segunda dimensión, cuando lo que realmente necesita añadir es simplemente otra matriz. Por ejemplo, regresando a nuestro programa de cricket, si el capitán del equipo quiere almacenar los nombres de los jugadores, no debería de añadir otra dimensión a la matriz de puntuaciones para realizar esto, en su lugar convendría declarar una segunda matriz de tipo cadena para almacenar el nombre de cada jugador:

```
int puntuaciones = new int[11];
string[] nombres = new string[11];
```

Asegúrese de mantener cruzados los nombres de cada jugador con su puntuación. En otras palabras, el elemento en el subíndice `0` de la matriz de nombres, debe contener la puntuación asociada a ese jugador en el elemento `0` de la matriz de puntuaciones.

Punto del Programador: Utilice matrices de pocas dimensiones

En todos mis años dedicados a la programación, nunca he tenido que utilizar una matriz de más de tres dimensiones. Si alguna vez se encuentra envuelto en un proyecto donde esté utilizando matrices con dimensiones excesivas, yo le sugeriría que pare su trabajo para volver atrás y reconsidere la manera en la que está solucionando el problema. Es probable que pueda obtener una solución mucho más eficiente creando una estructura, y posteriormente estableciendo una matriz con los elementos de la estructura. Vamos a hablar sobre las estructuras más adelante.

3.4 Errores y Excepciones

Odio cuando las cosas salen mal en mi programa. Si hay una situación que haga sentirse estúpido a un programador, es cuando alguien consigue que su programa falle de forma imprevista. Considere:

```
Introduzca su edad: Veintiuno
```

El programa pide al usuario que introduzca su edad. El usuario introduce su edad, pero lo hace sin utilizar dígitos numéricos. Lo que supone un contratiempo para su programa:

```
Unhandled Exception: System.FormatException: La cadena de entrada no tiene el
formato correcto. en System.Number.StringToNumber(String str, NumberStyles
options, NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo
info) en ProgramaDeRob.Main(String[] args)
```

Esto le coloca en una injusta situación de desventaja. Alguien lo suficientemente tonto como para escribir su edad con palabras, es capaz de interrumpir la ejecución del programa. En esta sección vamos a descubrir cómo los programadores de C# debemos controlar los errores y cómo hacer que nuestros programas sean mucho más difíciles de interrumpir.

3.4.1 Excepciones y el método Parse

`Parse` es el método que utilizamos para convertir cadenas de texto en valores numéricos.

```
int edad = int.Parse(cadenaEdad);
```

La sentencia anterior utiliza `int.Parse` para convertir la cadena de entrada `cadenaEdad`, en un resultado de tipo entero que posteriormente se almacena en la variable `edad`.

Esto funciona bien, pero no está exento de limitaciones. El problema del método `Parse` es que si le da una cadena que contiene texto no válido, éste no sabe qué hacer con ella. En el trabajo, si alguna vez tengo un problema al que no puedo hacer frente, intentaré resolverlo pasándoselo a mi jefe. Después de todo, mi jefe gana mucho más dinero y se le paga para solucionar imprevistos.

`Parse` solventa sus problemas lanzando una *excepción* para que otra parte del programa la capture. Esto es el equivalente en C# a escribir una nota al jefe que diga “No sé qué hacer con este inconveniente. ¿Alguna idea?” Si nadie captura la excepción (en efecto, el jefe no está cerca para hacer frente al problema), entonces la excepción hará finalizar el programa. El sistema de ejecución de C#, mostrará el contenido de la excepción y a continuación se detendrá, como sucedió en el ejemplo mostrado arriba.

3.4.2 Captura de Excepciones

Podemos hacer que nuestros programas lidien con entradas no válidas de texto manejadas por el método `Parse`, añadiendo código que *capture* las excepciones que el método `Parse` lanza, e intentar solucionar el problema. En términos de programación este proceso es denominado "*manejo de errores dinámicos*", ya que nuestro programa responde a un error cuando se produce. Para realizar esta tarea tendremos que utilizar una nueva construcción C#, la cláusula `try` – `catch`. La palabra reservada `try` es seguida por un bloque de código. Después del bloque de código aparece la cláusula `catch`. Si alguna de las instrucciones siguientes a la cláusula `try` lanza una excepción, el programa ejecutará el código existente en la cláusula `catch` para controlar este error.

```
int edad;
try
{
    edad = int.Parse(cadenaEdad);
    Console.WriteLine("Gracias");
}
catch
{
    Console.WriteLine("Valor de la edad, no válido");
}
```

Código de Ejemplo 17 Capturar Excepción simple

El código anterior utiliza el método `Parse` para decodificar la cadena `edad`. Sin embargo, la acción de analizar la cadena gramaticalmente tiene lugar dentro del bloque `try`. Si la llamada de `Parse` produjera una excepción, el código existente en el bloque `catch` se ejecutará y mostrará un mensaje de error al usuario. Tenga en cuenta que una vez la excepción ha sido lanzada, el código no retornará al bloque `try`, es decir, si el análisis gramatical de la cadena falla, el programa no mostrará el mensaje "`Gracias`".

3.4.3 El objeto Exception

Cuando le paso un problema a mi jefe, tendré que entregarle una nota que lo describa. Las excepciones trabajan de la misma manera. Una excepción es un tipo de objeto que contiene los detalles de un problema que se ha producido. Cuando `Parse` falla, se crea un objeto de excepción que describe el problema que acaba de ocurrir (en este caso la cadena de entrada no tiene el tipo correcto que se esperaba). El programa anterior ignora el objeto de excepción, y solamente registra al evento `exception`, aunque podemos mejorar el diagnóstico de nuestro programa capturando la excepción si así lo deseamos:

```
int edad;
try
{
    edad = int.Parse(cadenaEdad);
    Console.WriteLine("Gracias");
}
catch (Exception e)
{

 // Obtener el mensaje de error de la excepción
 Console.WriteLine(e.Message);
}
```

Código de Ejemplo 18 Obtener Mensaje de Excepción

La instrucción `catch` parece un método de llamada, con la excepcion `e` siendo un parámetro del método. Así es cómo funciona. Dentro de la cláusula `catch`, el valor de `e` se establece en la excepción que fue lanzada por `Parse`. El tipo `Exception`, tiene una propiedad denominada `Message` que contiene una cadena que describe el error. Si un usuario introduce una cadena no válida, el programa mostrará el texto interno de la excepción que contiene el mensaje, como vimos por ejemplo en el programa anterior:

```
La cadena no tiene el formato correcto.
```

Este mensaje se obtiene desde la excepción que fue lanzada. Esto es de utilidad si el código dentro del bloque `try`, pudiera lanzar varios tipos diferentes de excepciones, dado que le mensaje reflejaría lo que realmente sucedió. El objeto `Exception` también contiene otras propiedades que pueden ser útiles.

3.4.4 Excepciones anidadas

Cuando se genera una excepción, el sistema en tiempo de ejecución, que gestiona la ejecución de su programa dentro de la computadora, buscará la cláusula `catch`. Si su programa no contiene una instrucción para la excepción, ésta será capturada por una cláusula `catch` que forma parte del sistema en tiempo de ejecución. El código de esta cláusula `catch`, mostrará los detalles de la excepción y luego detendrá la ejecución del programa. Los programas pueden tener múltiples niveles `try – catch`, y el programa utilizará la cláusula `catch` que coincida con el nivel del código en el bloque `try`.

```
try
{
    // Las excepciones a este nivel serán capturadas por la
    // cláusula catch "externa"

    try
    {
        // Las excepciones a este nivel serán capturadas por la
        // cláusula catch "interna"
    }
    catch (Exception interna)
    {
        // Esta es la cláusula "interna"
    }

        // Las excepciones a este nivel serán capturadas por la
        // " cláusula catch "externa"
}
catch (Exception externa)
{
```

```
        // Esta es la cláusula catch "externa"
}
```

El código anterior muestra cómo funciona. Si el código del bloque más interno lanza una excepción, el sistema en tiempo de ejecución alcanzará la cláusula `catch` interna. Sin embargo, una vez que la ejecución abandone/a ese bloque interno, la cláusula externa es la que se ejecutará en caso de que se produzca una excepción.

Punto del Programador: No capture todas las excepciones

Ben llama a esto el síndrome Pokemon®: "Hazte con todas". No se sienta obligado a capturar todas las excepciones que el código pueda lanzar. Si alguien pide al método `Cargar` que busque un archivo para cargar que no existe, lo mejor que puede hacer es lanzar una excepción de tipo "archivo no encontrado", y no tratar de gestionarlo devolviendo una referencia nula o un elemento vacío. Al retornar un marcador de posición solamente conseguirá retrasar la aparición de problemas, cuando más adelante en alguna otra parte del código, se intente utilizar este elemento vacío que fue devuelto. Cuanto más rápido logre que un fallo aparezca, más fácil será identificar la causa que lo provoca.

3.4.5 Añadiendo una cláusula Finally

A veces hay acciones que su programa debe hacer independientemente de si una excepción es o no lanzada. Estas acciones incluyen tareas como cerrar archivos, liberar recursos, mostrar notificaciones al usuario o registrar algún dato en una base de datos. Sin embargo, sabemos que cuando una excepción es lanzada, las instrucciones de la cláusula `catch` correspondiente son ejecutadas, y el programa nunca retorna a la parte `try` del programa. El código de la cláusula `catch` podría regresar del método desde el que se está ejecutando o incluso lanzar una excepción propia. En estas situaciones, cualquier código que siga a la construcción `try – catch`, no tendrá la oportunidad de ejecutarse.

Afortunadamente, C# proporciona una solución a este problema permitiéndole añadir una cláusula `finally` a su construcción `try-catch`. Las instrucciones dentro de la cláusula `finally` se ejecutarán independientemente, de si el código dentro del bloque `try` lanza una excepción o no.

```
try
{
    // Código que podría lanzar una excepción
}
catch (Exception externa)
{
    // Código que captura la excepción
}
```

```
finally
{
    // Código que será ejecutado
    // se haya lanzado o no una excepción
}
```

En el código anterior, las instrucciones que se encuentran dentro del bloque `finally` serán ejecutadas con certeza, ya sea a la finalización de ejecutar el código del bloque `try`, o justo antes de que la ejecución abandone la parte `catch` de su programa.

3.4.6 Lanzar una Excepción

Ahora que sabemos cómo capturar excepciones, vamos a considerar cómo lanzarlas.

```
throw new Exception("Boom");
```

La instrucción anterior crea una nueva excepción, y la lanza a continuación. Al crear una nueva excepción, puede especificar una cadena que contenga el mensaje que describa el error. En este caso he utilizado el mensaje inútil "`Boom`". Al lanzar una excepción podría provocar que su programa finalizara, si no se ejecuta el código dentro de una construcción `try – catch`.

3.4.7 Etiqueta Exception

Es mejor reservar las excepciones para situaciones en las que su programa realmente no pueda continuar. Esto es una manera de afrontar cualquier situación inesperada o excepcional que se presente mientras se ejecuta un programa. Así pues, recuerde que solamente debería lanzar una excepción en las situaciones donde su programa no pueda realmente hacer otra cosa. Por ejemplo, si un usuario en el programa de doble acristalamiento introduce una altura de ventana demasiado grande, el programa puede simplemente imprimir el mensaje "Altura demasiado grande" y pedir otro valor. Lo que significa que los errores a este nivel, no deben lanzar excepciones.

Punto del Programador: Planifique el control y gestión de excepciones

Al diseñar un programa debe considerar qué eventos cuentan como "críticos (obstáculos insuperables)" y cómo va a lidiar con ellos. Además, debe valorar si es necesario crear sus propias excepciones personalizadas, basándose en las proporcionadas por el propio Sistema, y utilizarlas para controlar los errores del programa.

3.5 La construcción Switch

Ahora ya sabe prácticamente todo lo que necesita saber sobre cómo desarrollar un programa en C#. Probablemente esta afirmación le sorprenda, pero hay muy poco que saber acerca de la programación en sí. La mayor parte del resto de fundamentos de C#, se centra en hacer el negocio de la programación más sencillo. Un buen ejemplo de esto es la construcción `switch`.

3.5.1 Tomando decisiones múltiples

Supongamos que está refinando su programa de la ventana de doble acristalamiento, para permitir que su cliente pueda seleccionar entre una gama predefinida de ventanas. El menú principal del programa sería parecido a esto:

```
Introduzca el tipo de ventana:

1 = ventana abatible
2 = estándar
3 = puerta del patio
```

Su programa puede entonces calcular el coste de la ventana elegida, seleccionando el tipo e introduciendo el tamaño requerido. Cada método realizará las preguntas pertinentes al usuario y calculará el precio final del artículo.

Cuando vaya a escribir el programa, probablemente terminará teniendo un código similar a este:

```
static void ventanaAbatible()
{
    Console.WriteLine("Ventana Abatible");
}
static void ventanaEstandar()
{
    Console.WriteLine("Ventana Estándar");
}
static void puertaDelPatio()
{
    Console.WriteLine("Puerta del Patio");
}
```

Estos métodos son los que eventualmente tratarán con cada tipo de ventana. Hasta ahora, tan solo imprimirán por pantalla, el nombre del método desde el que han sido llamados. Poco a poco iremos completando el código restante (esta es realmente una manera efectiva de construir sus programas). Una vez que ya hemos realizado la parte inicial correspondiente a los métodos, lo siguiente que debemos de hacer es escribir el código que llamará a la opción seleccionada por el usuario.

Anteriormente, creamos un método que podemos utilizar para obtener un número por parte del usuario (este método denominado **leerEntero** utilizaba como parámetros un prompt y unos valores establecidos para los límites de altura y anchura de una ventana). Nuestro programa puede usar este método para obtener el valor elegido, y a continuación seleccionar el método que deba utilizar.

3.5.2 Selección de opciones utilizando la construcción if

Cuando tenga que realizar la selección, probablemente terminará con un código parecido a este:

```
int seleccion;
seleccion = leerEntero( "Tipo de ventana: ", 1, 3 );

if (seleccion == 1)
{
    ventanaAbatible();
}
else
{
    if (seleccion == 2)
    {
        ventanaEstandar();
    }
    else
    {
        if (seleccion == 3)
        {
            puertaDelPatio();
        }
        else
        {
            Console.WriteLine( "Número no válido" );
        }
    }
}
```

Esta solución funciona, pero es bastante chapucera. Al utilizar un gran número de construcciones `if`, hacen que este código no sea del todo óptimo.

3.5.3 La construcción switch

Debido a que tendrá que realizar este tipo de selecciones habitualmente, C# contiene una estructura de control especial que le permite seleccionar una opción entre varias posibles establecidas con un valor particular. Esta estructura de control especial, es denominada construcción *switch*. Si utiliza ésta para escribir el código anterior, su programa se vería así:

```
switch (seleccion)
{
    case 1:
          ventanaAbatible();
          break;
    case 2:
          ventanaEstandar();
          break;
    case 3:
          puertaDelPatio();
          break;
    default:
          Console.WriteLine( "Número no válido" );
          break;
}
```

Código de Ejemplo 19 Utilizar una construcción Switch

La construcción `switch` evalúa el valor de una variable que se pasa como argumento y que utiliza para decidir qué opción ejecutar. Ésta ejecuta la cláusula `case` que coincida con el valor de la variable pasada a la construcción `switch`. Esto significa que el tipo de los valores `cases` que utilice, deben ser del mismo tipo que el utilizado por la variable de selección `switch`. Después del método existente en cada evaluación, debe haber una instrucción `break` para salir del bloque y continuar la ejecución del programa en la instrucción siguiente a la construcción `switch`.

Otra cláusula muy útil es `default`, que debe situarse al final de todas las cláusulas `case`. Esta cláusula proporciona a la construcción `switch` una manera de ejecutar instrucciones en caso de que no se verifique ninguno de los casos evaluados; en nuestro caso, podemos imprimir por pantalla un mensaje de salida apropiado.

Puede utilizar la construcción `switch` con otros tipos diferentes a los numéricos, si así lo necesita:

```
switch (comando)
{
    case "abatible":
          ventanaAbatible();
          break;
    case "estandar":
          ventanaEstandar();
          break;
    case "patio":
          puertaDelPatio();
          break;
    default:
          Console.WriteLine( "Comando no válido" );
          break;
}
```

Esta construcción `switch`, utiliza una cadena de texto para controlar la selección de los casos. Sin embargo, sus usuarios no agradecerán que haga esto así, ya que ellos tendrían que escribir el nombre completo de la opción que deseen, y por supuesto si escribieran un carácter incorrecto, la elección no sería reconocida.

Multiples sentencias condicionales case

Puede utilizar múltiples sentencias condicionales `case` para que su programa pueda ejecutar una determinada instrucción `switch`, si el comando coincide con una de las varias opciones disponibles:

```
switch (comando)
{
    case "abatible" :
    case "a" :
          ventanaAbatible();
          break ;
    case "estandar" :
    case "e" :
          ventanaEstandar();
          break ;
    case "patio" :
    case "p" :
          puertaDelPatio();
          break ;
    default :
          Console.WriteLine( "Comando no válido" );
          break ;
}
```

La anterior construcción `switch`, seleccionará una opción concreta si el usuario introduce el nombre completo del comando o sus letras iniciales. Si desea realizar una selección basada en cadenas de texto como esta, le aconsejo revisar los métodos `ToUpper` y `ToLower` proporcionados por el tipo `string`. Estos métodos pueden ser utilizados para obtener una versión de una cadena que se encuentre totalmente en mayúsculas o minúsculas, lo que hace que la comprobación de los comandos sea mucho más sencilla de realizar:

```
switch (comando.ToUpper())
{
    case "ABATIBLE" :
    case "A" :
    ...
}
```

Punto del Programador: Las construcciones switch son muy útiles

Las construcciones `switch` hacen que un programa sea más fácil de entender, como también más rápido de escribir. Estas también hacen mucho más fácil añadir comandos adicionales, ya que bastará con añadir otra cláusula `case` al código. Sin embargo, le aconsejo que no ponga grandes fragmentos de código dentro de una construcción `switch - case`. En lugar de esto, es mejor utilizar una llamada a un método, como hemos visto anteriormente.

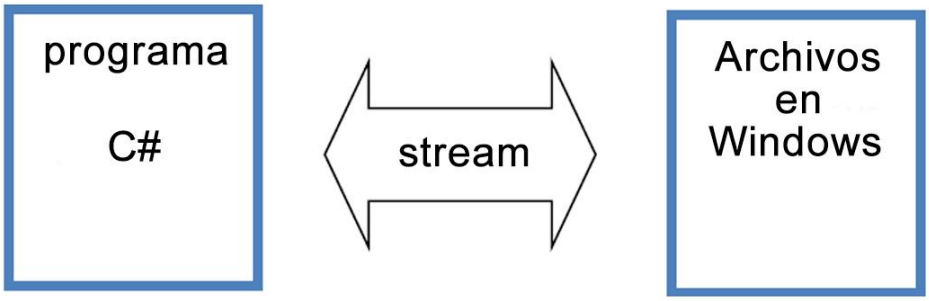
3.6 Utilizar Archivos

Si quiere que sus programas sean tremendamente útiles, tendrá que incorporarles la forma de que los datos queden almacenados una vez el programa deje de estar ejecutándose. Sabemos que podemos guardar datos en `archivos`, ya que normalmente es la forma en que los programas que utilizamos lo hacen.

Los archivos son manejados y gestionados por el sistema operativo de la computadora. Lo que nosotros queremos hacer es utilizar C#, para indicarle al sistema operativo que queremos crear archivos y que nos permita acceder a ellos cuando lo necesitemos. La buena noticia, es que, aunque los sistemas operativos difieren unos de otros en la forma que manejan y gestionan sus archivos, la forma en que se manipulan los archivos en C# es la misma para cualquier computadora. Podemos escribir un programa C# para crear un archivo en un PC con Windows, y luego utilizar ese mismo programa para crear un archivo en un sistema UNIX sin problemas.

3.6.1 Streams y Archivos

C# hace uso de un concepto genérico llamado *stream* que permite a los programas trabajar con archivos. Un stream es un vínculo entre su programa y un recurso de datos. Los datos son pensados como una transferencia de un punto al otro como un flujo de datos o información, de modo que pueden ser utilizados para leer y escribir en archivos. El stream es lo que vincula su programa con el sistema operativo de la computadora que está utilizando. El sistema operativo realmente hace el trabajo, y la biblioteca de C# que está utilizando es la que convierte su petición de utilizar streams en instrucciones para el sistema operativo que esté utilizando en ese momento:



Un programa C# puede contener un objeto que representa a un determinado stream que un programador ha creado y vínculado a un archivo. El programa realiza operaciones en el archivo realizando llamadas a los métodos pertenecientes al objeto stream para indicarle lo que debe hacer.

C# tiene diferentes objetos de flujo que podemos utilizar dependiendo de la tarea que tengamos que hacer. Todos ellos se utilizan exactamente de la misma manera. De hecho, ya está familiarizado con la forma en que se utilizan los streams, dado que la clase `Console`, que conecta un programa C# con el usuario, es implementada como un flujo (en este caso de entrada). Los métodos `ReadLine` y `WriteLine` son comandos que puede establecer en cualquier punto en el que se requiera leer y escribir datos.

Vamos a considerar dos clases de stream, que permiten a los programas utilizar archivos; estas son las clases `StreamWriter` y `StreamReader`.

3.6.2 Creación de un flujo de salida

Puede crear un objeto stream tal y como crearía cualquier otro, mediante el uso de la palabra reservada `new`. Cuando el flujo es creado podemos pasarle el nombre del archivo que debe ser abierto.

```
StreamWriter escribir;
escribir = new StreamWriter("prueba.txt");
```

La variable `escribir` se crea para hacer referencia a la instancia en la que desea escribir. `StreamWriter` es creado, se autoriza la apertura de un archivo en modo escritura llamado `prueba.txt`. Si este proceso falla por cualquier motivo, ya sea debido a que el sistema operativo no lo permite escribir en el archivo, o porque el nombre utilizado no es válido, será lanzada una excepción apropiada para el error producido.

Sin embargo, tenga en cuenta que este código no encontrará ningún problema si el archivo `prueba.txt` ya existe. Lo que ocurrirá en es este caso, es que se creará un nuevo archivo vacío reemplazando al existente. Esto es potencialmente peligroso. Lo que significa que utilizando las dos instrucciones anteriores podría destruir completamente el contenido de un archivo existente,

lo cual podría ser muy grave. La mayoría de los programas preguntan al usuario si debe o no sobrescribirse un archivo existente, descubrirá más adelante cómo hacer esto.

3.6.3 Escribir un flujo de datos

Una vez que el flujo ha sido creado, puede empezar a escribir datos en los archivos, llamando a los métodos de escritura que éste proporciona.

```
escribir.WriteLine("Hola mundo");
```

La instrucción anterior realiza una llamada al método `WriteLine` en el flujo de trabajo para que escriba el texto "`Hola mundo`" en el archivo `prueba.txt`. Esta es exactamente la misma técnica que es utilizada para escribir información en la consola para que el usuario la lea. De hecho, puede utilizar todas las características de escritura, incluyendo aquellas que aprendimos en la sección de dar formato al texto impreso por pantalla.

Cada vez que escriba una línea en el archivo, se añade al final del archivo de texto. Si su programa entra dentro de un bucle infinito mientras realiza el proceso de escritura, es posible que el sistema pueda quedarse sin espacio de almacenamiento. Si esto ocurre, y el proceso de escritura no se puede realizar con éxito, la llamada al método `WriteLine` lanzará una excepción. Un programa debidamente escrito, debería asegurarse que cualquier excepción como esta (también se pueden lanzar cuando se abre un archivo), sea capturada y gestionada correctamente.

3.6.4 Cerrar un flujo de datos

Cuando su programa ha terminado de escribir un flujo de datos, es muy importante que el flujo sea cerrado explícitamente utilizando el método `Close`:

```
escribir.Close();
```

Cuando el método `Close` es llamado, el flujo escribirá el texto que está a la espera de ser escrito en el archivo, y desconectará el programa del mismo. Cualquier intento posterior de escribir en el flujo de trabajo fallará, lanzando una excepción. Una vez que un archivo ha sido cerrado, éste puede entonces ser utilizado por otros programas existentes en la computadora, es decir, una vez realizado el cierre, puede utilizar el programa `Notepad` para abrir el archivo `prueba.txt` y revisar lo que hay en su interior. Olvidarse de cerrar un archivo es perjudicial por varias razones:

- Es posible que el programa finalice sin que el archivo esté cerrado correctamente. En esta situación, algunos de los datos que escribió en el archivo no estarán allí presentes.

- Si su programa tiene un flujo vinculado a un archivo, es posible que otros programas no puedan utilizar ese archivo. Esto hará imposible mover o renombrar el archivo.

- Un flujo abierto consume una parte pequeña, pero significativa, de recursos. Si su programa crea muchos flujos de trabajo, pero no los cierra, podría causar problemas al abrir otros archivos posteriormente.

Así pues, cierre los archivos o sufrirá las consecuencias.

3.6.5 Streams y Espacios de nombres

Si se apresura y prueba el código anterior, comprobará que no funciona. Siento que esto sea así. Hay algo más que necesita saber antes de poder utilizar el tipo `StreamWriter`. Al igual que muchos de los objetos que se ocupan de la entrada y salida, este objeto es definido en el sistema de *espacio de nombres* `System.IO`. Nosotros ya hemos hecho referencia a los espacios de nombres anteriormente, cuando reflejamos la necesidad de utilizar la sentencia `using System;` al inicio de nuestros programas en C#. Ahora necesitamos conocerlos un poco más en profundidad.

Los espacios de nombres se utilizan para la búsqueda de recursos. El lenguaje C# proporciona las palabras reservadas y construcciones que nos permiten escribir programas, pero por encima de esto hay una gran cantidad de recursos adicionales suministrados con una instalación C#. Estos recursos, son elementos como el objeto `Console` que nos permite leer y escribir texto. Una instalación C# realmente contiene miles de recursos, cada uno de los cuales debe ser identificado de manera única. Si estuviera a cargo de catalogar un gran número de elementos, encontraría muy útil agrupar los elementos. Los conservadores de museos hacen esto todo el tiempo. Ellos colocan todos los artefactos romanos en una sala, y los griegos en otra. Los diseñadores del lenguaje C # crearon un espacio de nombres, donde los programadores pueden hacer el mismo trabajo con sus recursos.

Un espacio de nombres es, literalmente, un “espacio donde los nombres tienen un significado”. El nombre completo de la clase `Console`, que ha estado utilizando para mostrarle texto por pantalla al usuario es `System.Console`. Es decir, la clase `Console` se encuentra en el espacio de nombres `System`. De hecho, se acepta utilizar esta forma completa en sus programas:

```
System.Console.WriteLine("Hola mundo");
```

La instrucción anterior utiliza el *nombre completo cualificado* del recurso de consola y llama al método `WriteLine` proporcionado por ese recurso. Sin embargo, no hemos tenido que utilizar este formato porque al inicio de nuestros programas, le hemos indicado al compilador que utilice el espacio de nombres `System`, para buscar cualquier nombre que no haya identificado antes. La directiva `using`, nos permite indicar al compilador donde buscar los recursos.

```
using System;
```

Esta sentencia indica al compilador que busque los recursos en el espacio de nombres `System`. Una vez que hemos especificado un espacio de nombres en un archivo de programa, ya no necesitamos indicar el nombre completo para los recursos de ese espacio de nombres. Siempre que el compilador encuentre un elemento que no ha identificado, éste automáticamente lo buscará en los espacios de nombres que hemos indicado que utilice. En otras palabras, cuando el compilador ve la sentencia:

```
Console.WriteLine("Hola mundo");
```

- busca en el espacio de nombres `System` ese objeto para que pueda ser utilizado el método `WriteLine`. Si el programador comete un error al escribir el nombre de la clase:

```
Consle.WriteLine("Hola mundo");
```

- el compilador buscará en el espacio de nombres `System`, un objeto llamado `Consle`, fallará y generará un error de compilación.

Este es el mismo error que obtendrá si intenta utilizar la clase `StreamWriter` sin indicar al compilador que lo busque en el espacio de nombres `System.IO`. En otras palabras, para utilizar las clases encargadas del manejo de archivos, necesitará añadir la siguiente declaración en la parte superior de su programa:

```
using System.IO;
```

Es posible poner un espacio de nombres dentro de otro (del mismo modo en que un bibliotecario pondría una vitrina de vasijas en la sala romana a la que podría referirse como Romanas.Vasijas), por eso el espacio de nombres `IO` se encuentra realmente dentro del espacio de nombres `System`. Sin embargo, solo porque utilice un espacio de nombres, no implica en que tenga que utilizar todos los espacios de nombres definidos dentro éste, por lo que debe incluir la línea anterior para que los objetos de manejo de archivos estén disponibles.

Los espacios de nombres son la forma ideal de asegurarse de que los nombres de los elementos que usted crea, no entren en conflicto con los de otros programadores. Veremos cómo crear sus propios espacios de nombres más adelante.

```
using System;
using System.IO;

class DemoEscrituraArchivo
{
    public static void Main()
    {
        StreamWriter escribir;
        escribir = new StreamWriter("prueba.txt");
        escribir.WriteLine("Hola mundo");
        escribir.Close();
        Console.ReadKey();
    }
}
```

Código de Ejemplo 20 Escritura de Archivos completa

3.6.6 Lectura de un archivo

La lectura de un archivo es muy similar a la escritura. Deberá declarar y crear un flujo para realizar el trabajo. En este caso, se utiliza la clase `StreamReader` para leer las líneas de información desde un archivo de texto estándar.

```
StreamReader leer = new StreamReader("prueba.txt");
string linea = leer.ReadLine();
Console.WriteLine(linea);
leer.Close();
```

El programa anterior vincula un flujo de trabajo al archivo `Prueba.txt`, lee la primera línea del archivo, lo muestra en pantalla y finalmente cierra el flujo. Si no se puede encontrar el archivo, entonces el intento de apertura fallará y el programa lanzará una excepción.

Detectar el final de un archivo de entrada

Repetidas llamadas al método `ReadLine` devolverá sucesivas líneas de un archivo. Sin embargo, si su programa llega al final del archivo, el método `ReadLine` devolverá una cadena vacía cada vez que sea llamado. Afortunadamente el objeto `StreamReader` proporciona una propiedad llamada `EndOfStream` que un programa puede utilizar para determinar cuándo se ha alcanzado el final del archivo. Cuando la propiedad se convierte en `true`, se ha alcanzado el final del archivo.

```
StreamReader leer = new StreamReader("prueba.txt");
while (leer.EndOfStream == false)
{
    string linea = leer.ReadLine();
    Console.WriteLine(linea);
}
leer.Close();
```

El código anterior abrirá el archivo `prueba.txt`, y mostrará cada línea del archivo en la consola. El bucle `while`, detendrá el programa cuando se alcance el final del archivo.

```
using System;
using System.IO;

class DemoEscrituraYLecturaArchivo
{
    public static void Main()
    {
        StreamWriter escribir;
        escribir = new StreamWriter("prueba.txt");
        escribir.WriteLine("Hola mundo");
        escribir.Close();

        StreamReader leer = new StreamReader("Prueba.txt");
        while (leer.EndOfStream == false)
        {
            string linea = leer.ReadLine();
            Console.WriteLine(linea);
        }
        leer.Close();
        Console.ReadKey();
    }
}
```

Código de Ejemplo 21 Escritura y Lectura de Archivo

El código del ejemplo anterior, inserta una línea de texto en un archivo y luego abre el archivo y lo imprime en pantalla.

3.6.7 Rutas de archivo en C#

Si ha utilizado una computadora durante un tiempo, estará familiarizado con el concepto de las carpetas (en ocasiones también denominados directorios). Éstas se utilizan para organizar la información que almacenamos en la computadora. Cada archivo que se crea se coloca en una carpeta determinada. Si utiliza Windows, encontrará que hay varias carpetas que el sistema crea para usted automáticamente. Una puede ser utilizada para Documentos, otra para Imágenes y otra

para Música. Usted puede crear sus propias carpetas dentro de estas (por ejemplo, Documentos\Artículos).

La ubicación de un archivo en un equipo es en ocasiones denominado la *ruta de acceso* al archivo. La ruta de acceso a un archivo puede ser dividida en dos partes, la ubicación de la carpeta y el nombre del archivo en sí. Si no especifica una ubicación de carpeta cuando abre un archivo (como hemos estado hacienda en los ejemplos vistos hasta ahora con el archivo `Prueba.txt`), el sistema asume que el archivo que se está utilizando debe guardarse en la misma carpeta en la que el programa se está ejecutando. En otras palabras, si está ejecutando el programa `LeerArchivo.exe` que se encuentra dentro de la carpeta `MisProgramas`, los programas anteriores asumirán que el archivo `Prueba.txt` debe guardarse también dentro la carpeta `MisProgramas`.

Si desea utilizar un archivo en una carpeta diferente (lo cual puede ser una buena idea, ya que los archivos de datos casi nunca se mantienen en el mismo lugar que los programas), puede añadir una ruta de acceso al nombre de archivo:

```
string Sistema;
ruta = @"c:\datos\2009\Noviembre\ventas.txt";
```

El código anterior crea una variable de tipo cadena, que contiene la ruta de acceso a un archivo llamado `ventas.txt`. Este archivo es guardado dentro de la carpeta Noviembre, que se encuentra a su vez almacenada dentro de la carpeta `2009`, que a su vez se encuentra dentro de la unidad `C`.

Los caracteres de barra invertida (\) en la cadena, sirven para distinguir y separar a las diferentes carpetas que forman la ruta del archivo. Tenga en cuenta que he especificado un literal de cadena que no contiene caracteres de control (que es lo que la `@` significa al inicio del literal), ya que de lo contrario los caracteres `\` en la cadena serán interpretados por C#, como el inicio de una secuencia de control. Si tiene problemas en donde sus programas no encuentran los archivos que usted sabe perfectamente que existen, le aconsejo que se asegure de que los separadores de ruta no se están utilizando como caracteres de control.

4 Creación de Soluciones

4.1 Nuestro caso de estudio: El Banco Amigo

La mayor parte de esta sección se basa en un caso de estudio, que le permitirá ver las características de C# en un contexto amplio y riguroso. Usted tomará el papel de un programador, que utilizará el lenguaje C# para crear una solución para un cliente.

El programa que vamos a desarrollar es para un banco, el "*Banco Bueno y Amigo de Personas Encantadoras ™*", también conocido como el *Banco Amigo*. Crearemos esta aplicación en C# para explorar las características del lenguaje que pueden ayudarnos a desarrollar de una manera más sencilla.

Es poco probable que llegue a implementar realmente un sistema bancario completo durante su carrera profesional como programador (aunque podría ser bastante divertido – y probablemente bastante lucrativo). Sin embargo, desde el punto de vista de la programación este es un problema interesante ya que su enfoque nos permitirá descubrir un montón de técnicas que nos será útil conocer para incorporarlas en otros programas que podríamos desarrollar.

Punto del Programador: Buscar patrones

El número de programas diferentes en el mundo es en realidad muy pequeño. Muchas de las operaciones que nuestra aplicación bancaria va a realizar (almacenar una gran cantidad de información sobre un gran número de individuos, buscar información sobre una persona en particular, implementar transacciones que modifican el contenido de uno o más elementos del sistema) son comunes a muchos otros tipos de programas, desde videojuegos a robots.

4.1.1 Alcance del Sistema Bancario

El *alcance* de un sistema es una descripción de las tareas que el sistema va a realizar. Esto es también, por implicación, una declaración de lo que el sistema **no** realizará. Esto es igualmente importante, ya que un cliente no suele tener una idea clara de lo que usted está haciendo y puede esperar que el programa haga cosas que usted no va a tener en cuenta a la hora de programarlo. Establecer el alcance del programa al inicio del proyecto, es de vital importancia para no encontrarse con sorpresas desagradables más adelante.

En este momento nos estamos limitando a gestionar la información de la cuenta bancaria. El gerente del banco nos ha dicho que el banco almacena información sobre cada cliente. Esta información incluye su nombre, dirección, número de cuenta, saldo y descubiertos. Otros datos de interés serán añadidos más adelante.

Existen miles de clientes y el gerente también nos ha informado que también hay un número de diferentes tipos de cuentas (y que de vez en cuando nuevos tipos de cuenta son ideados).

El sistema también debe generar cartas de aviso e informativas, según sea requerido.

4.2 Tipos enumerados

Esto suena realmente pijo. Si alguien le pregunta lo que ha aprendido hoy, puede contestarle "He aprendido a utilizar tipos enumerados", lo que causará gran asombro en su interlocutor. Bueno… claro, esto será así, a no ser que su interlocutor sea programador. Si se encuentra en este caso, probablemente él le responderá "¡Oh! eso significa que has enumerado algunos estados".

4.2.1 Enumeraciones y estados

Enumerado suena demasiado pijo. Pero si usted piensa en "enumerado" tan solo como "numerado", le resultará más sencillo entenderlo. Para comprender realmente lo que estamos haciendo aquí, tenemos que considerar el problema que estos tipos están destinados a resolver.

Sabemos que, para almacenar un valor entero, podemos utilizar una variable de tipo `int`. Para almacenar valores que sean verdaderos o falsos, podemos utilizar una variable de tipo `bool`. Sin embargo, a veces queremos almacenar un conjunto o grupo de valores o estados particulares.

Ejemplo de estados

Los tipos enumerados son muy útiles a la hora de almacenar informaciones de estado. Los estados no son otra cosa que el nombre del cliente o el saldo de su cuenta.

Por ejemplo, si estoy escribiendo un programa para jugar al juego *Battleships* (donde las cuadrículas del "mar" almacenan diferentes tipos de embarcaciones que pueden ser atacadas y hundidas), puedo determinar que en una cuadrícula de mar podría encontrar lo siguiente:

- Agua (cuadrícula vacía)
- Tocado
- Acorazado
- Crucero
- Submarino
- Embarcación

Si piensa en ello, estoy ensamblando más metadatos aquí, determinando los elementos del mar a los que debo hacer un seguimiento y registrando este tipo de información. También puede utilizar números, si lo prefiere:

- Agua = 1
- Tocado = 2
- Acorazado = 3
- Crucero = 4
- Submarino = 5
- Embarcación = 6

Sin embargo, esto significaría que tendría que hacer un seguimiento de los valores en sí, y por tanto tener que recordar que, si recibimos el valor 7 durante un turno de jugada, esto es evidentemente incorrecto.

C# incorpora una forma en la que podemos crear un tipo que tenga solamente un conjunto particular de valores posibles. Estos tipos son denominados "tipos enumerados":

```
enum EstadoMar
{
    Agua,
    Tocado,
    Acorazado,
    Crucero,
    Submarino,
    Embarcación
} ;
```

He creado un tipo llamado `EstadoMar` que puede ser utilizado para almacenar el estado de una zona particular del mar. Este solamente puede tener los valores indicados anteriormente, y deben ser manejados únicamente en el ámbito de estas enumeraciones designadas. Por ejemplo, debo escribir:

```
EstadoMar marAbierto;
marAbierto = EstadoMar.Agua;
```

Mi variable `marAbierto`, solamente puede almacenar valores que representan el estado del contenido del mar. Por supuesto, C# en sí, representará a estos estados como valores numéricos particulares, pero la manera en que se gestionen los estados internamente no es un problema del que deba preocuparme.

Tenga en cuenta que los tipos que creo (como `EstadoMar`), son resaltados por el editor en un color azul que no es exactamente del mismo tono que el utilizado para las palabras reservadas.

Esto demuestra que estos elementos son tipos adicionales que he creado, que se pueden utilizar para crear variables, pero en realidad no son parte del lenguaje C# en sí, como sí lo son las palabras reservadas. Es importante que comprenda lo que está sucediendo aquí. Anteriormente hemos utilizado tipos de datos que son parte de C#, por ejemplo, `int` y `double`. Ahora hemos

llegado al punto en el que realmente estamos creando nuestros propios tipos de datos, que pueden utilizarse para almacenar valores de datos que son requeridos por nuestra aplicación.

4.2.2 Creación de un tipo enumerado

El nuevo tipo `enum` puede ser creado fuera de cualquier clase, y crea un nuevo tipo que puede utilizarse en cualquiera de mis programas:

```
using System;

enum Semaforo
{
    Rojo,
    RojoAmbar,
    Verde,
    Ambar
};

class DemoEnum
{
    public static void Main()
    {
        Semaforo luz;
        luz = Semaforo.Rojo;
        Console.WriteLine(luz);
        Console.ReadKey();
    }
}
```

Código de Ejemplo 22 Enumerado Semáforo

Cada vez que tenga que almacenar algo que pueda tomar un número limitado de valores posibles, o estados (por ejemplo, `EnVenta`, `EnOferta`, `Vendido`, `Agotado`, etc.), debería pensar en utilizar tipos enumerados para almacenar los valores.

Punto del Programador: Utilizar tipos enumerados

Los tipos enumerados tienen beneficios importantes donde programas, programadores y usuarios se benefician si se utilizan correctamente. El programa se vuelve más fácil de escribir, más fácil de comprender y más seguro. Por lo tanto, debería utilizar este tipo de datos con asiduidad.

Para el software bancario, se necesita almacenar el estado de un elemento que registre la información sobre el cliente. Por ejemplo, podríamos tener los estados "Congelada", "Nueva", "Activa", "Cerrada" y "Bajo auditoría" como estados para nuestra cuenta bancaria. Si este es el caso, es razonable crear un tipo enumerado que pueda contener estos valores y no otros.

```
enum EstadoCuenta {
    Nueva,
    Activa,
    BajoAuditoria,
    Congelada,
    Cerrada
} ;
```

Ahora tenemos una variable que puede contener información del estado de una cuenta en nuestro banco. Cada cuenta contendrá una variable de tipo `EstadoCuenta`, que representa el estado de esa cuenta.

4.3 Estructuras

Las estructuras nos permiten organizar una serie de valores individuales dentro de un conjunto de datos, que podemos asignar a uno de los elementos del problema en el que estamos trabajando. Esto es importante en muchas aplicaciones.

4.3.1 ¿Qué es una estructura?

Frecuentemente cuando esté tratando con información querrá almacenar una colección de diferentes elementos relacionados con un asunto. El Banco Amigo le ha encargado un sistema de almacenamiento de cuentas, y usted puede utilizar estructuras para hacer esto más fácilmente. Usted debería empezar haciendo lo siguiente:

1. Establecer con precisión la especificación, es decir, obtener exactamente de forma escrita lo que esperan que su sistema haga.

2. Negociar unos honorarios exorbitantes.

3. Considerar cómo va a realizar el almacenamiento de datos.

Una estructura de muestra

De su especificación usted conoce que su programa debe almacenar lo siguiente:

- nombre del cliente – tipo cadena (string)
- dirección del cliente – tipo cadena (string)
- número de cuenta – valor de tipo entero (integer)
- saldo de la cuenta - valor de tipo entero (integer)
- limite descubierto - valor de tipo entero (integer)

El Banco Amigo le ha indicado que solamente necesitan introducir hasta 50 personas en el sistema de almacenamiento bancario. Después de un rato usted regresa con lo siguiente:

```
const int MAX_CLIENTES = 50;

EstadoCuenta[] estados = new EstadoCuenta [MAX_CLIENTES];
string[] nombres = new string [MAX_CLIENTES];
string[] direcciones = new string [MAX_CLIENTES];
int[] NosCuentas = new int [MAX_CLIENTES];
int[] saldos = new int [MAX_CLIENTES];
int[] descubierto = new int [MAX_CLIENTES];
```

Lo que tenemos, es una matriz para cada dato que necesitamos almacenar de cada cliente. Si estuviésemos hablando de una base de datos (que en realidad es lo que estamos desarrollando), el volumen de datos de cada cliente sería un registro de cada parte individual de ese volumen, por ejemplo. el valor de descubierto, sería un campo. En nuestro programa estamos trabajando sobre la base de que `saldos[0]` almacena el saldo del primer cliente de nuestra base de datos, `descubierto[0]` almacena el descubierto del primer cliente, y así sucesivamente. (Recuerde que los subíndices de las matrices empiezan en 0).

Esto está muy bien, ya que podría desarrollar un sistema de base de datos con esta estructura de datos. Sin embargo, sería mucho mejor poder agrupar su registro de datos de una manera más definida.

4.3.2 Creación de una estructura

C# permite crear estructuras de datos. En C#, una estructura es una colección de variables que necesitan ser tratadas como una sola entidad. En C# a un conjunto de datos se le denomina una estructura, y cada una de las partes de esta es denominado un campo. Para facilitar nuestro trabajo con nuestra base de datos bancaria, podríamos crear una estructura que pudiera almacenar toda la información sobre un cliente:

```
struct Cuenta
{
    public EstadoCuenta Estado;
    public string Nombre;
    public string Direccion;
    public int NumeroCuenta;
    public int Saldo;
    public int Descubierto;
};
```

Esto define una estructura, llamada `Cuenta`, que contiene toda la información requerida del cliente. Una vez hecho esto, podemos definir algunas variables:

```
Cuenta CuentaDeRob;
```

Esta sentencia crea una única variable `Cuenta`, denominada `CuentaDeRob`. El programa puede establecer valores en los elementos miembros de esta cuenta.
Podemos referirnos a miembros individuales de una estructura, escribiendo su nombre después de la variable de estructura que estamos usando con un `.` (punto) de separación entre ambos elementos, por ejemplo:

```
CuentaDeRob.Nombre="Rob";
```

- la instrucción anterior asigna el valor `Rob`, en la variable `Nombre` de la estructura `CuentaDeRob`.

```
using System;

enum EstadoCuenta
{
    Nueva,
    Activa,
    BajoAuditoria,
    Congelada,
    Cerrada
} ;

struct Cuenta
{
    public EstadoCuenta Estado;
    public string Nombre;
    public string Direccion;
    public int NumeroCuenta;
    public int Saldo;
    public int Descubierto;
} ;
```

```
class ProgramaBancario
{
    public static void Main()
    {
        Cuenta CuentaDeRob;
        CuentaDeRob.Estado = EstadoCuenta.Activa;
        CuentaDeRob.Saldo = 1000000;
    }
}
```

Código de Ejemplo 23 Estructura de Cuenta generosa

El anterior fragmento de código crea una variable de estructura de tipo `Cuenta` a la que se ha denominado `CuentaDeRob`, a continuación, establece el estado de la cuenta a `Activa`, y finalmente le otorga un millón (aunque no hace nada con este dinero).

Observe cómo la estructura y la enumeración han sido declaradas fuera de la clase `Programa`Bancario. Una vez que he creado mi estructura, puedo utilizarla de la misma manera que si fuera un tipo `int` o `float`.
Las estructuras son realmente útiles cuando creamos matrices de ellas. Como vimos en el programa de anotaciones de los jugadores de cricket, se hace difícil gestionar elementos individuales, pero todo se vuelve más fácil si utilizamos una matriz.

```
const int MAX_CLIENTES = 100;
Cuenta[] Banco = new Cuenta [MAX_CLIENTES];
```

Estas instrucciones crean una matriz íntegra de clientes, llamada `Banco`, destinada a almacenar todos los clientes. Tenga en cuenta que inteligentemente he establecido una variable llamada `MAX`_CLIENTES, destinada actualmente para poder almacenar hasta `100` clientes.

Podemos asignar una variable de estructura a otra, tal y como haríamos con cualquier otro tipo de variable. Cuando la asignación es realizada, todos los valores en la estructura de origen son copiados en la estructura de destino:

```
Banco[0] = CuentaDeRob;
```

Esta instrucción copiará la información de la estructura CuentaDeRob, en el elemento inicial de la matriz `Banco`.

También puede hacer esto con elementos de una matriz de estructuras, de esta forma:

```
Banco[25].Nombre
```

- sería la cadena que contiene el nombre del cliente en el elemento con subíndice `25`.

4.3.3 Utilización de una estructura

Una vez que tengamos nuestra estructura podemos utilizarla para almacenar los datos de nuestros clientes bancarios:

```
class MatrizEstructuraCuenta
{

    public static void Main()
    {
        const int MAX_CLIENTES = 100;
        Cuenta[] Banco = new Cuenta[MAX_CLIENTES];
        Banco[0].Nombre = "Rob";
        Banco[0].Estado = EstadoCuenta.Activa;
        Banco[0].Saldo = 1000000;
        Banco[1].Nombre = "Jim";
        Banco[1].Estado = EstadoCuenta.Congelada;
        Banco[1].Saldo = 0;
   }

}
```

Código de Ejemplo 24 Establecer Información de Cuenta en Matrices

Puede ver cómo funcionaría esto en el ejemplo de código anterior, que crea una matriz de 100 registros de clientes, y luego establece los dos primeros elementos de la matriz. ¿Cuál de los titulares de la cuenta tiene más dinero?

Este programa no lee los datos de los 100 clientes del banco, pero sí le muestra cómo tener acceso a los distintos campos de una matriz en una estructura.

Valores iniciales en estructuras

Cuando una estructura es creada como una variable local (es decir en un bloque), los valores en ésta son indefinidos. Esto significa que si intenta utilizarlas en su programa obtendrá un error de compilación. Esto es exactamente lo mismo, que, si utilizara una variable, antes de haberle otorgado un valor. En otras palabras:

```
Cuenta CuentaDeRob;
Console.WriteLine( "Su Nombre es: " + CuentaDeRob.Nombre );
```

- produciría un error de compilación. Su trabajo como programador es asegurarse de establecer un valor en las variables de su programa, antes de realizar cualquier operación donde tenga que utilizarlas.

Utilización de Tipos Estructura en llamadas a Métodos

Un método puede tener parámetros de tipo estructura:

```
public void ImprimirCuenta(Cuenta a)
{
    Console.WriteLine("Nombre: " + a.Nombre);
    Console.WriteLine("Dirección: " + a.Direccion);
    Console.WriteLine("Saldo: " + a.Saldo);
}
```

Este método proporciona una forma rápida de imprimir los contenidos de una variable de cuenta:

```
ImprimirCuenta(CuentaDeRob);
```

El valor de la estructura `CuentaDeRob`, es pasado al método para que trabaje con estos datos, al igual que en nuestros programas anteriores de ejemplo, suministrábamos valores de tipo entero. Es posible pasar los elementos de la matriz en la llamada al método (ya que un elemento que forma parte de la matriz de valores `Cuenta`, es por definición una instancia de `Cuenta`).

```
class ProgramaBancario
{
    public static void ImprimirCuenta(Cuenta a)
    {
        Console.WriteLine ("Nombre: " + a.Nombre);
        Console.WriteLine ("Dirección: " + a.Direccion);
        Console.WriteLine ("Saldo: " + a.Saldo);
   }

    public static void Main()
    {
        const int MAX_CLIENTES = 100;
        Cuenta[] Banco = new Cuenta[MAX_CLIENTES];
        Banco[0].Nombre = "Rob";
        Banco[0].Direccion = "Domicilio de Rob";
        Banco[0].Estado = EstadoCuenta.Activa;
        Banco[0].Saldo = 1000000;
        ImprimirCuenta(Banco[0]);
        Banco[1].Nombre = "Jim";
        Banco[1].Direccion = "Domicilio de Jim";
        Banco[1].Estado = EstadoCuenta. Congelada;
        Banco[1].Saldo = 0;
        ImprimirCuenta(Banco[1]);
        Console.ReadKey();
   }
}
```

Código de Ejemplo 25 Imprimir Valores de Cuenta desde una Matriz

También es posible crear métodos que retornen resultados que sean de un tipo estructura, por ejemplo, podríamos crear un método `LeerCuenta` que lea una cuenta y retorne sus valores.

Punto del Programador: Las estructuras son cruciales

En un sistema comercial, es bastante común dedicar mucho tiempo al diseño de las estructuras para el almacenamiento de datos. Éstas son los pilares del programa, ya que contienen todos los datos sobre los que se construye todo lo demás. Puede considerar el diseño de estructuras y su contenido, como otra gran cantidad de metadatos que debe recopilar referentes al sistema que debe crear.

Diseño con Tipos

Puede observar que hemos añadido un valor `Estado` a la estructura `Cuenta`. Con esto conseguimos que sea más fácil realizar un seguimiento del estado particular de una cuenta. Lo que hemos hecho es crear un tipo que pueda contener una serie de valores (nuestro tipo enumerado `EstadoCuenta`), e implementarlo dentro de otro tipo que hemos diseñado para almacenar la información del titular de la cuenta bancaria. Este enfoque es el que usted debe utilizar a la hora de abordar problemas más complejos. Podemos obtener todos los detalles necesarios que deben contener los tipos, de los metadatos recogidos y recopilados en la conversación que mantuvimos con nuestro cliente.

Por ejemplo, si queremos crear un impresionante programa para gestionar ventanas de doble acristalamiento, podríamos desarrollar un código como este:

```
enum EstadoVentana
{
    Presupuestada,
    Encargada,
    Fabricada,
    Enviada,
    Instalada
};

struct Ventana
{
    public EstadoVentana Estado;
    public double Anchura;
    public double Altura;
    public string Descripcion;
};
```

Este código podría almacenar la información sobre una ventana de una casa, inclyendo las dimensiones de la ventana, el estado de la ventana encargada y una descripción con todos los detalles de la ventana. Para una casa que disponga de varias ventanas, podríamos crear una matriz de estructura `Ventana`.

Punto del Programador: Debe otorgar un estado a sus objetos

La clase `Cuenta` vista anteriormente, es un buen ejemplo de un objeto que puede ser utilizado de diferentes formas, dependiendo del estado en que se encuentre. Por ejemplo, no será posible retirar fondos de una cuenta que esté en el estado `Congelada`. Casi todos los objetos de un sistema, pueden ser perfeccionados mediante la adición de un estado. Los pedidos de productos pueden estar EnPreparacion, `EnEsperaDePago`, `Cancelados`, `Entregados` o `Reembolsados`. Los aliens pueden `Dormir`, `Atacar` o bien haber sido `Exterminados`. Siempre que invente un nuevo objeto debe inmediatamente comenzar a considerar los estados que puede ocupar en su sistema.

4.4 Objetos, Estructuras y Referencias

Ya ha visto que, si necesita almacenar un bloque de información sobre un determinado elemento, puede reunir todos los detalles del mismo, en una estructura. Las estructuras son útiles, pero nosotros queremos resolver otro tipo de problemas que se presentan cuando escribimos programas más grandes:

- Queremos asegurarnos de que un determinado elemento de nuestro programa, no pueda ser accidentalmente establecido a un estado no válido. Es decir, nosotros no queremos tener cuentas bancarias con números de cuenta vacíos o incorrectos.

- Queremos dividir un sistema grande y complejo en componentes distintos y separados, que puedan ser desarrollados independientemente y ser intercambiados con otros que realicen el mismo trabajo. Es decir, nosotros queremos que un equipo de programadores trabaje con las cuentas, otro equipo con los cheques, otro con las tarjetas de crédito etc.

- Queremos asegurarnos de que el esfuerzo destinado a la creación de nuevos tipos de cuentas bancarias sea lo más pequeño posible. Es decir, si el banco decidiera introducir una nueva cuenta de depósito de alto interés, queremos poder hacer uso de la cuenta de depósito ya existente.

Para poder conseguir todo esto, vamos a tener que empezar a considerar los programas desde un punto de vista orientado al diseño de objetos. Esta sección debería venir acompañada con algún tipo de advertencia sobre la salud, del tipo "algunas de estas consideraciones podrían ocasionarle dolores de cabeza en la fase inicial", no obstante, los siguientes puntos son muy importantes:

- Los objetos no añaden nuevas funcionalidades a los programas – como indiqué anteriormente, ya hemos aprendido todos los aspectos esenciales a la hora de desarrollar programas: asignaciones, bucles condicionales, matrices, etc.

- Los objetos son considerados como una solución a nivel de diseño. Estos nos permiten hablar sobre sistemas en términos generales. Podemos volver atrás y refinar cómo los objetos realizan sus tareas posteriormente.

Podríamos desarrollar cualquier programa existente, utilizando las técnicas que hasta ahora hemos aprendido, pero los objetos nos permiten poder trabajar de una manera mucho más lógica y eficaz. Por lo tanto, es conveniente que utilicemos este paradigma, sea o no de nuestro agrado…

4.4.1 Objetos y Estructuras

En C#, los objetos y las estructuras tienen muchas características en común. Ambas pueden almacenar datos y contener métodos. Sin embargo, hay una diferencia crucial entre ambas. Las estructuras son gestionadas en términos de *valor*, mientras que los objetos son gestionados en términos de *referencia*.

Es muy importante que entienda las diferencias existentes entre ambas, ya que esto tiene un gran impacto en la forma en que se utilizan.

Creación y uso de una Estructura

Considere el siguiente código:

```
struct EstructuraCuenta
{
    public string Nombre;
};

class DemoEstructurasYObjetos
{
    public static void Main()
    {
        EstructuraCuenta EstructuraCuentaDeRob;
        EstructuraCuentaDeRob.Nombre = "Rob";
        Console.WriteLine(EstructuraCuentaDeRob.Nombre);
        Console.ReadKey();
    }
}
```

Código de Ejemplo 26 Estructura de Cuenta simple

Este código implementa una cuenta bancaria muy simple, donde solamente almacenamos el nombre del titular de la cuenta. El método `Main` crea una variable de estructura, a la que hemos llamado `EstructuraCuentaDeRob`.

EstructuraCuentaDeRob
Nombre: Rob

A continuación, se establece la propiedad nombre de la variable en la cadena "Rob". Tal y como esperamos, si ejecutáramos este programa, se imprimiría por pantalla el nombre "Rob". Si la estructura contuviese otros elementos sobre la cuenta bancaria, estos también serían almacenados en la estructura, y podría utilizarlos de la misma manera.

Crear y Utilizar una Instancia de Clase

Podemos hacer una pequeña modificación en nuestro programa, y convertir la cuenta bancaria en una clase:

```
class Cuenta
{
    public string Nombre;
};

class DemoEstructurasYObjetos
{
    public static void Main()
    {
        Cuenta CuentaDeRob;
        CuentaDeRob.Nombre = "Rob";
        Console.WriteLine(CuentaDeRob.Nombre);
    }
}
```

Código de Ejemplo 27 Clase Cuenta no compilable

La información de la cuenta está ahora contenida en una clase, en lugar de en una estructura. La clase de la cuenta es denominada simplemente, `Cuenta`. El problema del código escrito anteriormente, es que cuando compilemos el programa, obtendremos el siguiente error:

```
Program.cs(12,3): error CS0165: Uso de la variable local no asignada
Cuenta 'CuentaDeRob'
```

Así pues, ¿qué está ocurriendo? Para entender lo que está sucediendo es necesario analizar el programa línea a línea:

```
Cuenta CuentaDeRob;
```

La instrucción anterior podría parecer una declaración de una variable denominada `CuentaDeRob`. Pero en el caso de los objetos, esto no es lo que parece.

Lo que realmente se produce cuando el programa atiende esta línea de código, es la creación de una *referencia* denominada `CuentaDeRob`. Tales referencias permiten *referenciar* a instancias de la `Cuenta`. Puede pensar en ellas como una etiqueta de equipaje, en las que pueden estar atadas a algo con un trozo de cuerda. Si dispone de la etiqueta, puede seguir el recorrido de la cuerda y llegar hasta el objeto al que está atada.
Pero cuando creamos una referencia, en realidad no recibimos el elemento a la que ésta se refiere. El compilador sabe esto, y emite un error ya que la línea de código:

```
CuentaDeRob.Nombre = "Rob";
```

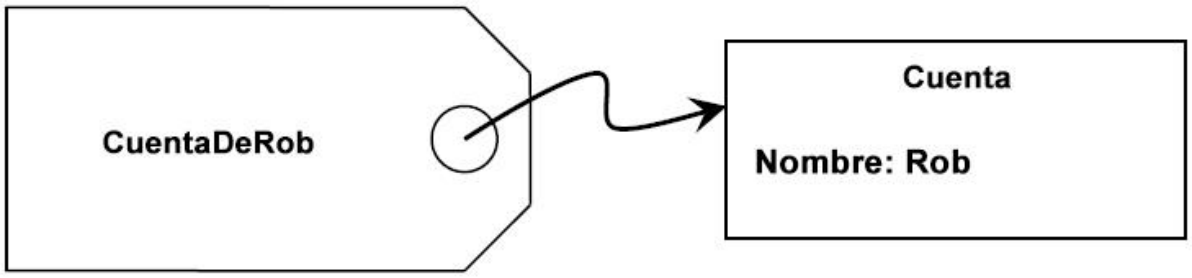
`-` intenta encontrar el elemento que está atado a esta etiqueta, y establecer la propiedad `Nombre` a "Rob". Dado que la etiqueta no está actualmente vinculada a ningún elemento de nuestro programa, fallará en este punto. El compilador por lo tanto se dice así mismo, “en efecto, se está tratando de seguir una referencia que no se refiere ni lleva hacía ningún elemento, así que voy a mostrarle un error de 'variable indefinida'".

Resolveremos el problema creando una instancia de la clase, y vinculando a continuación nuestra etiqueta a ésta. Esto se logra añadiendo una línea de código a nuestro programa:

```
class Cuenta
{
    public string Nombre;
};

class DemoEstructurasYObjetos
{
    public static void Main()
    {
        Cuenta CuentaDeRob ;
        CuentaDeRob = new Cuenta();
        CuentaDeRob.Nombre = "Rob";
        Console.WriteLine (CuentaDeRob.Nombre);
        Console.ReadKey();
    }
}
```

Código de Ejemplo 28 Clase Cuenta compilable

La línea de código que he añadido, crea un nuevo objeto `Cuenta` y establece la referencia `CuentaDeRob` para referirse a éste.



Ya hemos visto anteriormente la palabra reservada new. Nosotros la utilizamos para crear matrices. Esto se debe a que una matriz se implementa realmente como un objeto, y es por ello que debemos usar la palabra reservada new para crearlas. El elemento que new crea es un *objeto*. Un objeto es una instancia de una clase. Voy a repetir esto último utilizando una fuente elegante:

"Un objeto es una instancia de una clase"

Si he vuelto a repetirlo es porque quiero que entienda que esto es muy importante. Una clase proporciona las instrucciones a C# sobre lo puede y debe hacer. La palabra reservada new desencadena que C# utilice la información de la clase para realmente crear una instancia. Observe que, en el diagrama anterior, he llamado al objeto Cuenta, no `CuentaDeRob`. Esto es debido a que la instancia del objeto no tiene el identificador `CuentaDeRob`, es simplemente a lo que `CuentaDeRob` está unido en este momento.

4.4.2 Referencias

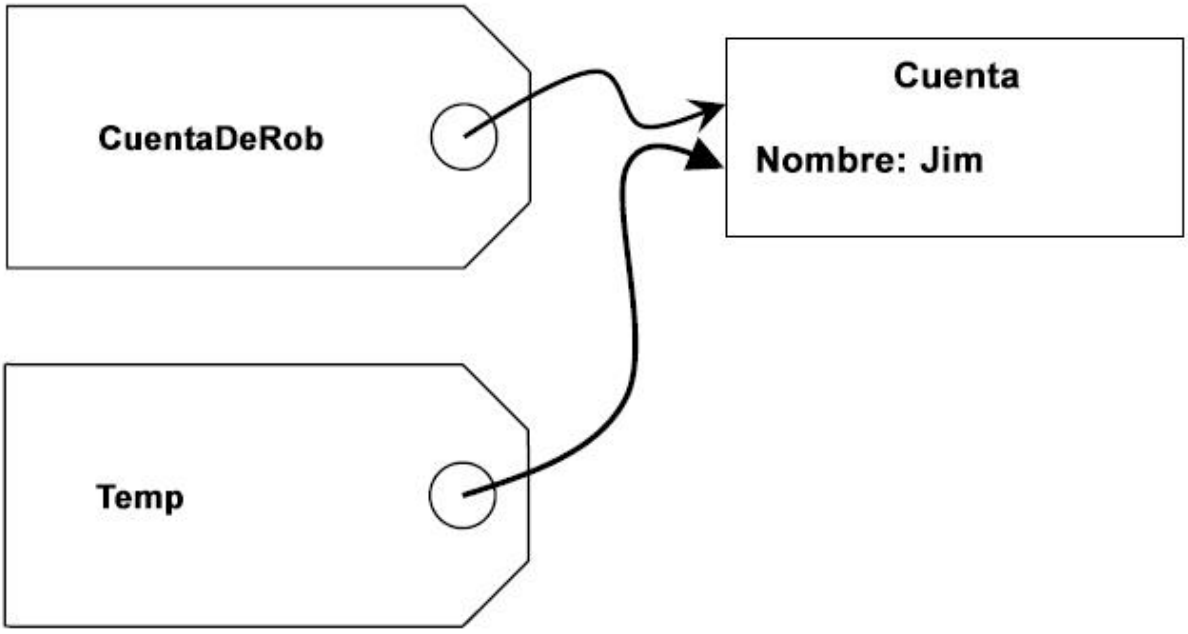
Ahora nos tenemos que acostumbrar a la idea de que, si queremos utilizar objetos, tendremos que hacer uso de las referencias. Estos dos conceptos vienen de la mano y son inseparables. Las estructuras son especialmente útiles, pero para cumplir con los requerimientos del diseño orientado a objetos tenemos que disponer de un objeto, lo que significa que debemos gestionar el acceso a un objeto en particular, haciendo uso de sus referencias. Realmente, esto no es tan doloroso, ya que casi todo el tiempo podemos tratar a la referencia como si realmente fuera el objeto en sí, pero debemos recordar que cuando mantenemos una referencia, no tenemos una instancia, sino que tenemos una etiqueta que está atada a una instancia.

Múltiples Referencias a una Instancia

Tal vez otro ejemplo de referencias, podría ser de ayuda en este punto. Considere el siguiente código:

```
Cuenta CuentaDeRob;
CuentaDeRob = new Cuenta();
CuentaDeRob.Nombre = "Rob";
Console.WriteLine(CuentaDeRob.Nombre);
Cuenta Temp;
Temp = CuentaDeRob;
Temp.Nombre = "Jim";
Console.WriteLine(CuentaDeRob.Nombre);
Console.ReadKey();
```

Código de Ejemplo 29 Referencias múltiples



Ambas etiquetas se refieren a la misma instancia de `Cuenta`. Así que cualquier cambio que realice al objeto al que hace referencia (apunta) `Temp`, se verá reflejado en el objeto al que también apunta `CuentaDeRob`, *porque son el mismo objeto*. Esto significa que el programa imprimirá `Jim`, dado que es el nombre almacenado en el objeto al que `CuentaDeRob` hace referencia.

Esto indica una dificultad añadida a los objetos y referencias. No hay ningún límite para el número de referencias que puedan ser asociadas a una instancia, por lo que debe recordar que al cambiar el objeto al que apunta una referencia, puede cambiar esa instancia desde el punto de vista de otros objetos.

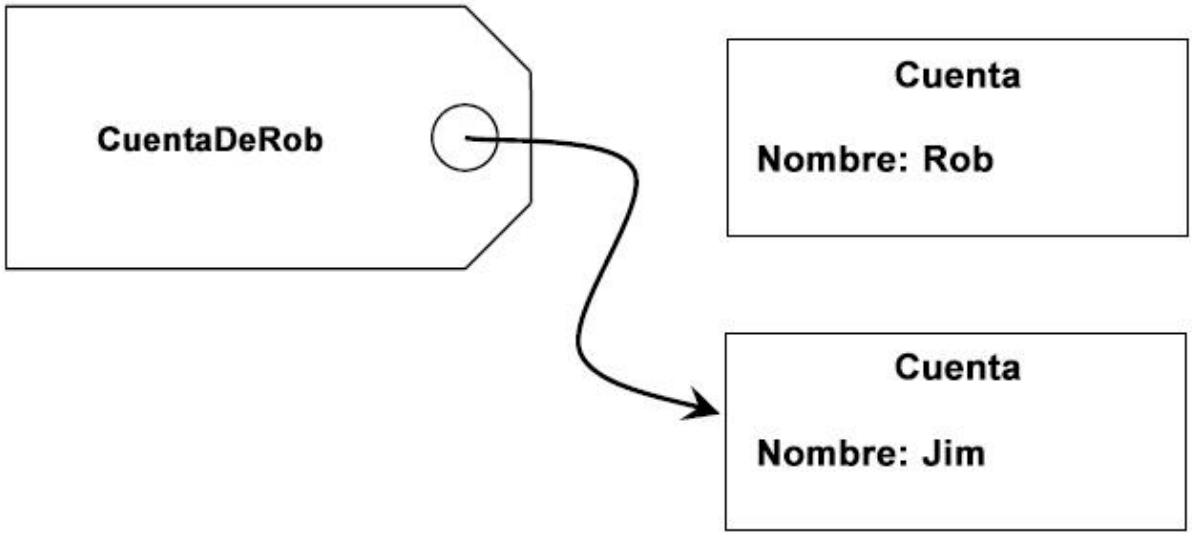
Sin referencias a una Instancia

Para completar la confusión, vamos a considerar lo que sucede cuando no existe ninguna referencia al objeto:

```
Cuenta CuentaDeRob;
CuentaDeRob = new Cuenta();
CuentaDeRob.Nombre = "Rob";
Console.WriteLine(CuentaDeRob.Nombre);
CuentaDeRob = new Cuenta();
CuentaDeRob.Nombre = "Jim";
Console.WriteLine(CuentaDeRob.Nombre);
```

Código de Ejemplo 30 Sin Referencias a una Instancia

El código anterior crea una instancia de cuenta, establece la propiedad de nombre de la misma a `Rob`, y luego crea otra instancia de cuenta. La referencia `CuentaDeRob` apunta al nuevo elemento, que tiene el nombre establecido a `Jim`. La pregunta es: ¿Qué ocurre con la primera instancia? Una vez más, esto se ve más claro a través de un diagrama:



La primera instancia se muestra "colgando" en el espacio, sin ninguna referencia apuntando hacia ella. En lo que respecta al desperdicio de memoria producido en este caso, sería mejor que esa referencia que ya no apunta a ninguna estancia ya no estuviese disponible. De hecho, el lenguaje C# implementa un proceso especial, llamado "Recolector de basura (Garbage Collector)", cuyo trabajo está destinado a encontrar elementos inservibles y deshacerse de ellos. Tenga en cuenta que el compilador no nos impedirá "descolgar" y por tanto deshacernos de elementos inservibles como este.

También debe recordar que puede obtener un efecto similar, cuando una referencia a una instancia sale fuera del ámbito en la que ha sido definida:

```
{
    Cuenta localVar;
    localVar = new Cuenta();
}
```

La variable `localVar` es local al bloque. Cuando la ejecución del programa abandona el bloque, la variable local es destruida. Esto significa que la única referencia a la cuenta también se elimina, lo que le otorga otro trabajo al recolector de basura.

Punto del Programador: Procure evitar utilizar el Recolector de Basura

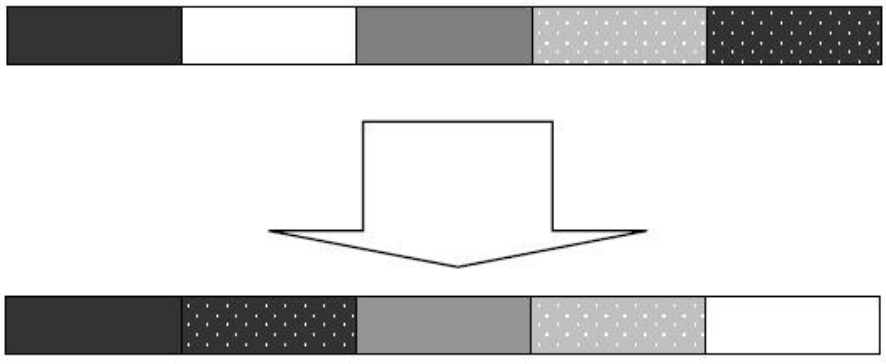
Aunque a veces es razonable liberar elementos, que no va a volver a utilizar, debe recordar que la creación y eliminación de objetos consumirá recursos del sistema. Cuando trabajo con objetos, me preocupa la cantidad de creaciones y destrucciones que estoy haciendo. Sólo porque los objetos sean eliminados automáticamente, no significa que deba abusar de esta ventaja.

4.4.3 ¿Por qué complicarse la vida utilizando Referencias?

Las referencias no parecen muy divertidas en este momento. Parecen más difíciles de crear y utilizar que los objetos, y pueden crear mucha confusión. Entonces, ¿por qué nos tenemos que complicar la vida utilizándolas?

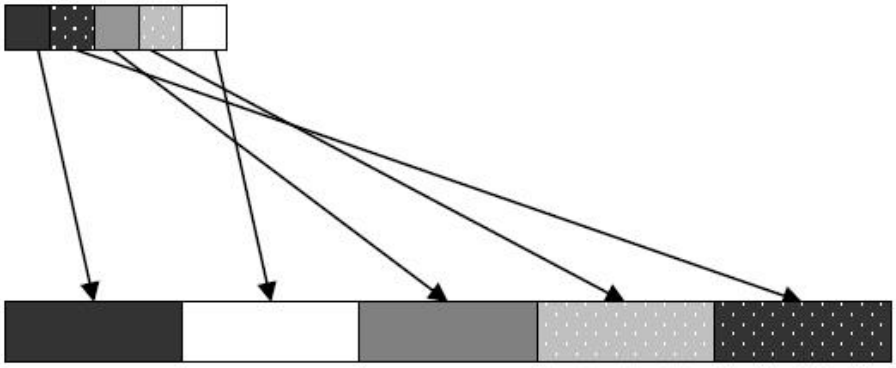
Para responder a esta cuestión, podemos considerar a la isla Yap del Pacífico. La moneda en uso en esta isla se basa en piedras de 12 pies de altura, que pesan varios cientos de libras cada una de ellas. El valor actualmente de una "moneda" en Yap, está directamente relacionado con el número de hombres que murieron en el barco que llevaba la roca a la isla. Cuando usted paga a alguien con una de estas monedas, realmente no la traslada físicamente y se la da a esa persona. En lugar de esto, usted le dice: "La moneda situada en la carretera de la cima de la colina es ahora tuya.". En otras palabras, los habitantes de la isla Yap, utilizan referencias para gestionar objetos que no quieren mover.

Esta es la razón por la que nosotros utilizamos referencias en nuestros programas. Considere un banco que contenga muchas cuentas de cliente. Si quisiéramos ordenarlas en orden alfabético por el nombre del cliente, tendríamos que moverlas todas, una a una.



Si nosotros mantuviéramos las cuentas como una matriz de elementos de estructura, tendríamos que trabajar demasiado solamente para mantener la lista ordenada. El banco también podría querer

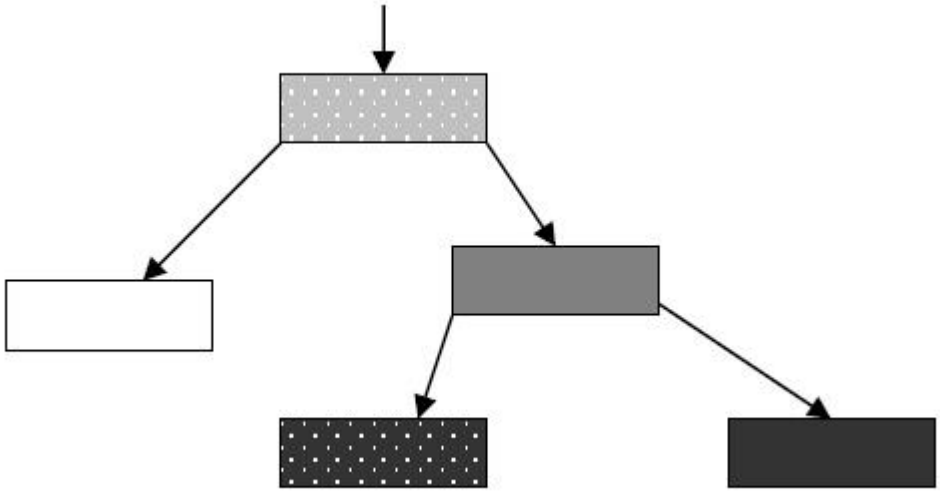
ordenar la información de varias maneras diferentes, por ejemplo, por el apellido del cliente y por el número de cuenta. Sin referencias, esto sería imposible. Con referencias, tan solo necesitamos mantener un número de matrices de referencias, cada uno de ellas ordenadas de una manera en particular:



Ordenando las referencias, nos evitamos tener que mover grandes cantidades de elementos de información. Podemos, además, añadir nuevos objetos sin tener que mover ningún objeto, ya que podemos mover las referencias en su lugar.

Referencias y Estructuras de Datos

Nuestra lista de referencias ordenadas es muy válida, pero si queremos añadir algún elemento nuevo a nuestra lista ordenada, tendríamos que desplazar las referencias. Podemos solventar esto, y también acelerar la búsqueda, estructurando nuestros datos en forma de árbol.



En el árbol superior, cada nodo tiene dos referencias; una puede apuntar a un nodo que es más "claro", la otra a un nodo que es más "oscuro". Si yo quiero una lista ordenada de los elementos, tengo que ir/dirigirme hasta el lado más "claro" que me sea posible, finalizando por tanto en el lado de mayor claridad. A continuación, iré al que se encuentra situado arriba del todo (que es el siguiente más claro). Posteriormente, bajaré por el lado más oscuro (Luke, yo soy tu padre), y repetiré el proceso. Uno de los aspectos más interesantes de este planteamiento, es también que añadir nuevos elementos es muy sencillo; Tan solo necesito encontrar el lugar en el árbol en el que tengo que insertar la referencia.

El proceso de búsqueda es también muy rápido, ya que puedo examinar cada nodo y decidir en cuáles de los caminos mirar a continuación, hasta que encuentre lo que estoy buscando o hasta que no haya ninguna referencia en la dirección deseada, en cuyo caso el elemento no se encuentra en la estructura.

Punto del Programador: Las Estructuras de Datos son importantes

Este no es un documento dedicado a las estructuras de datos, este es un documento de programación. Si no entiende todavía todo lo concerniente a los árboles, no se preocupe. Tan solo recuerde que las referencias son un mecanismo importante para crear estructuras de datos y por ahora es suficiente con esto. En el futuro, necesitará crear estructuras siguiendo este planteamiento.

La importancia de las Referencias

La clave de esta forma de trabajar es que un objeto puede contener referencias a otros objetos, así como a la carga útil de los datos (*payload* en inglés). Consideraremos este aspecto del uso de objetos más adelante; por ahora, solo necesitamos recordar que una referencia y un objeto son distintos e independientes.

NOTAS BANCARIAS: REFERENCIAS Y CUENTAS

Para una entidad bancaria que cuenta con miles de clientes, el uso de las referencias es crucial para tratar con los datos almacenados. Las cuentas se almacenan en la memoria de la computadora y debido al tamaño de cada cuenta y al número de cuentas que se tendrán que guardar, no será posible moverlas para ordenarlas desde la propia memoria principal.

Esto significa que la única manera de manipularlas, es dejarlas en el mismo lugar y tener una lista de referencias hacía ellas. Las referencias son pequeñas "etiquetas", que pueden ser utilizadas para localizar al elemento real en memoria. Ordenar una lista de referencias es muy sencillo, dando además la posibilidad de tener varias listas de este tipo disponibles para presentarlas de manera inmediata. Lo que significa que podemos ofrecer al director gerente una vista de su banco ordenada por nombres de cliente, y otra vista ordenada por el saldo disponible en cada cuenta. Y si el director gerente viene algún día con la necesidad de tener una nueva estructura o vista, también podremos crearla a través de referencias.

4.5 Diseño con Objetos

Ahora vamos a empezar a pensar en términos de objetos. La razón por la que lo hacemos, es porque siempre tendremos que hacer el diseño de nuestros sistemas tan sencillo como sea posible. Todo esto viene a cuento de la famosa “pereza creativa” por la que son conocidos todos los programadores. Lo que tratamos de hacer aquí se expresa mejor como:

“Dejar todo el trabajo difícil para más tarde, y si es posible busca algo que lo haga por ti.”

Los objetos nos permiten hacer esto. Si regresamos a nuestra cuenta bancaria, podemos ver que hay una serie de cosas que necesitamos poder hacer con ella:

- ingresar dinero en la cuenta
- sacar dinero de la cuenta
- conocer el saldo disponible en la cuenta
- imprimir un estado de cuenta
- cambiar la dirección del titular de la cuenta
- imprimir la dirección del titular de la cuenta
- cambiar el estado de la cuenta
- conocer el estado de la cuenta
- cambiar el límite autorizado para operar en descubierto
- detectar y conocer el límite en descubierto

En lugar de decir “Debemos poder efectuar estas operaciones en una cuenta bancaria”, el diseño basado en objetos pone patas arriba este pensamiento en tu cabeza, tal y como el presidente Kennedy hizo en sus años de mandato:

"Y así, mis compatriotas americanos: no os preguntéis qué puede hacer vuestro país por vosotros, preguntaos qué podéis hacer vosotros por vuestro país" (vítores y aplausos)

Nosotros no hacemos cosas para la cuenta bancaria. En lugar de ello, le pedimos a ésta que haga estas cosas para nosotros. El diseño de nuestra aplicación bancaria puede ser interpretado en términos de identificar los objetos que vamos a utilizar para representar la información, para posteriormente especificar qué cosas deben ser capaces de hacer. La parte realmente astuta es que una vez hemos decidido lo que la cuenta bancaria debe hacer, nosotros tenemos que buscar algo que realice este trabajo.

Si nuestra especificación es adecuada y está implementada correctamente, no tenemos que preocuparnos precisamente sobre quienes realizan este trabajo – nosotros solo tenemos que sentarnos y atribuirnos el mérito del trabajo realizado.

Esto nos lleva de nuevo a considerar un par de temas recurrentes en este documento; *metadatos* y *pruebas de software (*también denominado *testing)*. Lo que un objeto de cuenta bancaria deba ser capaz de hacer, forma parte de los metadatos de ese objeto. Una vez que hayamos decidido las acciones que la cuenta debe realizar, lo siguiente que debemos hacer es diseñar una forma en la que cada una de las acciones puedan ser probadas o testeadas.

En esta sección, vamos a implementar un objeto que tenga algunos de los comportamientos apropiados para una cuenta bancaria.

Punto del Programador: No Todo Debe Ser Posible

Tenga en cuenta que también hay algunas acciones que **no** deberíamos poder realizar con nuestros objetos de la cuenta bancaria. El número de cuenta identificativo de una cuenta es algo que es exclusivo de esa cuenta y no debe cambiar nunca. Podemos conseguir este comportamiento simplemente no proporcionando ningún método para que pueda ser modificado. Es importante reconocer en tiempo de diseño, las acciones que no deberían ser posibles efectuar, junto con las que sí se deberían poder realizar. Nosotros podríamos incluso identificar como algunas cosas han sido auditadas, ya que un objeto mantendrá un seguimiento de lo que se ha hecho con él. De esta forma, podemos averiguar fácilmente si hay cosas que se están haciendo mal.

4.5.1 Datos en Objetos

Así pues, podemos considerar nuestra cuenta bancaria en términos de lo que queremos que haga por nosotros. Lo primero que debemos identificar son todos los elementos de datos que queremos almacenar en ella. En aras de una mayor simplicidad, por ahora, solamente voy a considerar cómo controlar el saldo de las cuentas Esto me permitirá describir todas las técnicas requeridas sin perderme en excesivos detalles.

```
class Cuenta
{
    public decimal Saldo;
}
```

La clase `Cuenta` contiene el miembro de datos que guarda el saldo de nuestras cuentas bancarias. Los miembros de una clase que almacenan un valor que definen algún dato o información son denominados *propiedades*. He utilizado el tipo de datos `decimal` para almacenar el saldo de las cuentas, dado que este tipo está especialmente diseñado para almacenar valores financieros.

Hemos visto que cada uno de los elementos de una clase es un *miembro* de ésta y es almacenado como parte de ella. Cada vez que creo una instancia de la clase, obtengo también todos los miembros de ella. Ya hemos comprobado que es muy fácil crear una instancia de una clase y establecer el valor de un miembro:

```
Cuenta CuentaDeRob;
CuentaDeRob = new Cuenta();
CuentaDeRob.Saldo = 99;
```

La razón por la que este código funciona es porque todos los miembros del objeto son *públicos* por lo que es posible acceder a ellos desde cualquier clase. Esto significa que cualquier programador involucrado en el desarrollo de la aplicación puede hacer cosas como:

```
CuentaDeRob.Saldo = 0;
```

`-` y dejarme sin dinero. Si queremos impedir que esto suceda, necesitamos proteger los datos que se encuentran dentro de nuestros objetos.

4.5.2 Protección de miembros dentro de los objetos

Si los objetos van a ser útiles, tenemos que tener una forma de proteger los datos existentes almacenados dentro de ellos. Idealmente, yo quiero tomar el control cuando alguien intente cambiar un valor de mis objetos, y detener el cambio si no me parece oportuno, antes de que se produzca. La palabra pija que utilizamos para esto es *encapsulación*. Quiero ocultar todos los datos importantes dentro de mi objeto, para tener un control completo sobre lo que se está haciendo. Esta tecnología es la clave de mi *programación defensiva*, un enfoque que está orientado a asegurar en

un programa el comportamiento esperado ante cualquier situación de uso por incorrecta o imprevisible que esta pueda parecer.

Por ejemplo, en nuestra aplicación bancaria, queremos asegurarnos que el saldo de la cuenta nunca pueda ser modificado de una manera que no podamos controlarlo. La primera medida que debemos tomar es la de evitar que el mundo exterior pueda modificar el valor del saldo:

```
class Cuenta
{
    private decimal Saldo;
}
```

La propiedad ya no está definida como `public`. Ahora está definida como `private`. Esto significa que las clases externas ya no tienen acceso directo a esta propiedad. Si escribo el código:

```
CuentaDeRob.Saldo = 0;
```

- Obtendré un mensaje de error al intentar compilar el programa:

```
DemoPrivado.cs(13,3): error CS0122: 'MiembroPrivado.Cuenta.Saldo' no es
accessible debido a su nivel de protección
```

El valor del saldo se encuentra ahora contenido dentro del objeto y no se encuentra disponible para el mundo exterior.

Cambiar miembros privados

Seguro que adivino lo que está pensando en este momento. Usted está pensando "¿Qué sentido tiene hacerlo privado? ahora no se puede cambiar el valor". Bueno, gracias por su voto de confianza, amigo mío. Resulta que sí que puedo cambiar el valor, pero solamente utilizando un código que realmente se ejecute en la clase. Considere el programa:

```
class Cuenta
{
    private decimal saldo = 0;

    public bool RetirarEfectivo(decimal cantidad)
    {
        if (saldo < cantidad)
        {
           return false;
        }
        saldo = saldo - cantidad;
        return true;
    }
}

class Banco
{
    public static void Main()
    {
        Cuenta CuentaDeRob;
        CuentaDeRob = new Cuenta();
        if (CuentaDeRob.RetirarEfectivo(5))
        {
           Console.WriteLine( "Efectivo retirado correctamente" );
        }
        else
        {
           Console.WriteLine ( "Fondos insuficientes" );
        }
        Console.ReadKey();
    }
}
```

Código de Ejemplo 31 Retirada de Efectivo Fondos insuficientes

Este código crea una cuenta y luego intenta extraer cinco libras de ella. Este código por supuesto fallará, ya que el saldo inicial en mi cuenta es cero, pero muestra cómo proporcionar acceso a los miembros de una cuenta. El método `RetirarEfectivo` es un miembro de la clase `Cuenta` y por lo tanto puede acceder a miembros privados de la clase.

Punto del Programador: Los metadatos crean Miembros y Métodos

No he hecho ninguna mención sobre los metadatos desde hace por lo menos cinco minutos. Tal vez ahora sea un buen momento para hacerlo. Los metadatos que yo he recopilado sobre mi sistema bancario, me guiarán a la hora de saber qué acceso dar a los miembros de mis clases. En el código anterior, la manera en la que protejo el valor del saldo de la cuenta, es un reflejo de cómo el cliente quiere que me asegure de que este valor sea gestionado correctamente.

Métodos públicos

Se habrá dado cuenta, que yo declaré de tipo public el método `RetirarEfectivo`. Esto significa que el código que se ejecuta fuera de la clase puede realizar llamadas a ese método. En este caso tiene que ser así, ya que nosotros necesitamos que la gente interactúe con nuestros objetos llamando a los métodos disponibles en ellos. En general, las reglas son:

- Si es un miembro de datos (es decir, contiene datos) de la clase, hazlo private
- Si este es un método (es decir, hace algo), hazlo public

Por supuesto, las reglas pueden romperse en ocasiones especiales. Si no le preocupa la posible corrupción de los miembros, y quiere que su programa se ejecute lo más rápido posible, puede declarar public un miembro de datos. Si quiere escribir un método que sólo se utilice dentro de una clase y realice alguna tarea especial confidencial o secreta puede declararlo private.

Punto del Programador: Utilice convenciones de codificación para mostrar cuales de los elementos son privados

Si examina atentamente mi código (y le aconsejo que realice esta buena práctica), habrá comprobado que cuando escribo el nombre de un elemento público, utilizo una letra mayúscula al inicio de su nombre (como ocurre en el caso de `RetirarEfectivo` para retirar dinero en metálico de nuestra cuenta bancaria). Sin embargo, la primera letra de los miembros privados las escribo en minúscula (como ocurre en el caso del miembro de datos del saldo de nuestra cuenta bancaria). Esto hace que mi código sea fácil de leer para cualquier persona, dado que se puede apreciar a partir del nombre del miembro de la clase si éste es público o privado. Esta convención también se aplica a las variables locales de un bloque de código. Estas (por ejemplo, la omnipresente `i`) siempre comienzan con una letra minúscula.

Otros programadores van más allá, hacen cosas como las de poner `m_` delante de los nombres de las variables que son miembros de una clase. Ellos nombrarían, por ejemplo, `m_saldo`, a la variable que almacena el valor del saldo, para que los miembros de la clase sean fáciles de identificar. Personalmente, yo no llego tan lejos, porque considero que con el nombre del miembro es normalmente suficiente, pero las opiniones difieren unas de otras. Lo más importante en estas situaciones, es que todo el equipo de programación adopte las mismas convenciones. De hecho, la mayoría de las empresas de desarrollo cuentan con documentos que establecen las convenciones de codificación que utilizan, y esperan que los desarrolladores se adhieran a estas.

4.5.3 Una Clase Cuenta completa

Ahora podemos crear una clase de cuenta bancaria que controle el acceso al valor del saldo:

```
public class Cuenta
{
    private decimal saldo = 0;

    public bool RetirarEfectivo(decimal cantidad)
    {
        if ( saldo < cantidad )
        {
           return false;
        }
        saldo = saldo - cantidad;
        return true;
    }

    public void IngresarEfectivo (decimal cantidad)
    {
        saldo = saldo + cantidad;
    }

    public decimal ObtenerSaldo()
    {
        return saldo;
    }
```

La clase de la cuenta bancaria que acabo de crear tiene el comportamiento que se debe esperar en un algoritmo bien desarrollado. He creado tres métodos que puedo utilizar para interactuar con un objeto de cuenta. Puedo ingresar efectivo, saber si hay saldo en la cuenta y retirar efectivo, por ejemplo:

```
Cuenta prueba = new Cuenta();
prueba.IngresarEfectivo(50);
```

Al final de este conjunto de instrucciones, la cuenta `prueba` debe tener un saldo de 50 libras. Si no es así, mi programa está defectuoso. El método `ObtenerSaldo` es lo que en programación se conoce como un *descriptor de acceso*, ya que permite acceder a los datos de mi objeto de negocio. Podría escribir un poco más de código para poner a prueba estos métodos:

```
Cuenta prueba = new Cuenta();
prueba.IngresarEfectivo(50);
if ( prueba.ObtenerSaldo() != 50 )
{
      Console.WriteLine("Prueba de Ingreso de efectivo fallida");
}
else
{
      Console.WriteLine("Prueba de Ingreso de efectivo correcta");
}
```

Código de Ejemplo 32 Testear la Clase Cuenta

Mi programa ahora se autotestea a sí mismo, ya que realiza una tarea, y justo a continuación, se asegura de que la operación se ha realizado. No obstante, todavía tendría que leer el resultado de todas las pruebas que yo quiera realizar, lo cual es un trabajo demasiado tedioso. Más tarde consideraremos el uso de *pruebas unitarias* que hacen este proceso mucho más sencillo.

Punto del Programador: Haga sonar una sirena cuando las pruebas fallen

La prueba realizada anteriormente es buena, pero puede ser mejorada. Esta solamente muestra un pequeño mensaje si la prueba falla. En este caso para notificar el fallo se tiene que leer el mensaje que aparece en pantalla, y si se nos pasa por alto se puede pensar que la aplicación funciona correctamente. Mis pruebas cuentan la cantidad de errores que se han producido. Si el total de errores encontrados es mayor que cero, se imprime un enorme mensaje de texto intermitente en color rojo para indicar que algo malo sucedió. Algunos equipos de desarrollo realmente conectan sirenas y luces rojas intermitentes a sus sistemas de prueba, por lo que es imposible que se pase por alto una prueba fallida. A continuación, buscan al programador que causó el fallo, y le hacen pagar el café de cada desarrollador que forma parte del equipo durante una semana.

4.5.4 Desarrollo Guiado por Pruebas

Yo amo el desarrollo controlado por pruebas. Puede apostar, que, si yo añado alguna nueva línea de código a mi programa, lo haré desde el enfoque del desarrollo guiado por pruebas. Este resuelve los siguiente tres problemas que describo a continuación:

1. Usted no realiza las pruebas al final del proyecto. Este suele ser el peor momento para realizarlas, ya que podría estar utilizando un código que escribió hace algún tiempo. Si los errores (bugs) se encuentran en un fragmento de código antiguo, tiene que realizar el esfuerzo de recordar cómo funcionaba ese código. Es mucho mejor probar el código mientras lo escribe, ya que es cuando se tiene la mejor comprensión posible de lo que se supone que ese código debe hacer.

2. Puede escribir código temprano en el proyecto que probablemente será de utilidad más adelante. Muchos proyectos están condenados al fracaso porque la gente comienza a programar antes de tener una comprensión adecuada del problema. Escribir las pruebas primero, es realmente una buena forma de refinar la comprensión. Y existe una buena posibilidad de que las pruebas que escriba sean útiles también en algún momento determinado.

3. Cuando corrija errores (bugs en inglés) en su programa, tiene que asegurarse de que esas correcciones que realice no quebranten el código en alguna otra parte del programa (una de las formas más comunes de introducir nuevos errores en un programa, se produce a la hora de corregir un error o bug existente). Si cuenta con un conjunto de pruebas automáticas establecidas que se ejecutan después de cada error corregido, ya tiene una forma de evitar que esto suceda.

Así que, por favor, desarrolle realizando pruebas. Me dará las gracias más adelante.

Punto del Programador: Algunas cosas son difíciles de probar

El desarrollo guiado por pruebas es una práctica de ingeniería que permite garantizar que el software cumple con los requisitos que se han establecido, pero este no resuelve todos los problemas. Hay ciertos tipos de programas que son realmente difíciles de poner a prueba de esta forma. Cualquier pantalla que presente una interfaz donde se espere que un usuario introduzca comandos y se espere una respuesta por parte de la computadora es muy complicado de testear de esta forma, porque, aunque es fácil enviar información a un programa, con frecuencia es mucho más difícil de ver como el programa responde. Mi enfoque en este tipo de situaciones es hacer que la interfaz de usuario sea parte de la capa de presentación y se encuentre situada más arriba de la capa de negocio y de las capas de datos que son las que reciben las solicitudes para hacer el trabajo. En el caso de nuestra cuenta bancaria, el código que proporciona la interfaz de usuario donde el cliente introduce la cantidad de efectivo a retirar, será muy simple y estará conectado directamente a los métodos proporcionados por mis objetos. Si mis objetos pasan las pruebas, yo puedo tener la seguridad de que la interfaz de usuario también está bien diseñada. Este modelo de desarrollo de software es conocido como *programación por capas*.

El otro tipo de programa que es muy difícil de testear de esta manera es un videojuego. Esto se debe principalmente a que la jugabilidad no es un aspecto para el que se pueda diseñar una prueba específica de control de calidad. Solamente cuando alguien dice “El juego es demasiado difícil porque no se puede matar al enemigo final de fase sin perder todas las vidas.” Puedes realmente hacer algo al respecto.
La única solución en esta situación es establecer muy claramente lo que sus pruebas están tratando de demostrar (para que los testadores humanos lo revisen y sepan a qué atenerse), y una vez comprobado, realizar los cambios oportunos para hacerlo más fácil. Por ejemplo, debería ser sencillo ajustar el daño que causa el impacto de un alien a la nave espacial controlada por el jugador, y la velocidad a la que se mueven todos los objetos del videojuego.

4.6 Elementos estáticos

Hasta el momento, todos los miembros que hemos creado en nuestra clase han sido parte de una instancia de la clase. Esto significa que cada vez que creamos una instancia de la clase `Cuenta`, nosotros generamos un miembro de `saldo`. Sin embargo, también podemos crear miembros que formen parte de la clase, es decir, existen fuera de cualquier instancia particular.

4.6.1 Miembros de clase estáticos

La palabra reservada `static` nos permite crear miembros que no son mantenidos en una instancia, sino en la propia clase.

Es muy importante que aprenda lo que `static` significa en el contexto de un programa desarrollado en C#. Lo hemos utilizado en casi todos los programas que hemos escrito:

```
class CuentaDePrueba
{
    public static void Main()
    {
        Cuenta prueba = new Cuenta();
        prueba.IngresarEfectivo(50);
        Console.WriteLine ("Saldo:" + prueba.ObtenerSaldo());
    }
}
```

La clase `CuentaDePrueba` cuenta con un método `static` denominado `Main`. Sabemos que este es el método que se llama para iniciarla ejecución del programa. Éste forma parte de la clase `CuentaDePrueba`. Si yo creo cincuenta instancias de `CuentaDePrueba`, todas ellas compartirán el mismo método `Main`. En términos de C#, la palabra reservada `static` declara a un miembro como parte de la clase, **no** como parte de una instancia de la clase. Repetiré esto de nuevo utilizando una fuente más pija, ya que esto es importante:

“Un miembro estático es un miembro de la clase, no un miembro de una instancia de la clase”

Yo no tengo que crear una instancia de la clase `CuentaDePrueba`, para poder utilizar el método `Main`. Así es como mi programa funciona, ya que cuando éste se inicia no hay creada ninguna instancia, y por eso este método **debe** estar presente, ya que de lo contrario el programa no podría ser inicializado.

`Static` no significa "no se puede cambiar". Espere, creo que es el momento de utilizar una fuente más pija:

Static no significa “no se puede cambiar”.

Los miembros de una clase declarados como `static`, pueden ser utilizados como cualquier otro miembro de una clase. Los miembros de datos y los métodos pueden ser declarados `static`.

4.6.2 Utilizar los miembros de datos estáticos de una clase

Quizás un ejemplo de dato `static`, pueda ser de ayuda en este punto. Considere los tipos de interés de nuestras cuentas bancarias. El cliente nos ha dicho que uno de los miembros de la clase cuenta, tendrá que almacenar el tipo de interés de las cuentas. En el programa podemos implementar este dato, añadiendo otro miembro a la clase que mantiene la tasa de interés actual:

```
public class Cuenta
{
    public decimal Saldo;
    public decimal TipoDeInteresCobrado;
}
```

Ahora, ya puedo crear cuentas y asignarles saldos y tipos de interés. (por supuesto, si yo tuviera que crear esto como es debido, designaría estos miembros de datos privados, y posteriormente proporcionaría los métodos para acceder a ellos, etc., pero por ahora lo haré lo más simple posible).

```
Cuenta CuentaDeRob = new Cuenta();
CuentaDeRob.Saldo = 100;
CuentaDeRob. TipoDeInteresCobrado = 10;
```

El inconveniente es que me han comunicado que la tasa de interés se mantiene para todas las cuentas. Si el tipo de interés cambiara, este debe ser cambiado en **todas** las cuentas. Esto significa que, para implementar el cambio, tendría que ir cuenta por cuenta actualizando la tasa. Esto sería tedioso, y si por un descuido olvidara una cuenta, posiblemente caro.

Resuelvo el problema haciendo el miembro de datos de la tasa de interés `static`:

```
public class Cuenta
{
    public decimal Saldo;
    public static decimal TipoDeInteresCobrado;
}
```

La tasa de interés es ahora parte de la clase, no parte de ninguna instancia. Esto significa que tengo que cambiar la manera en la que accedo a esta propiedad:

```
Cuenta CuentaDeRob = new Cuenta();
CuentaDeRob.Saldo = 100;
```

```
CuentaDeRob.TipoDeInteresCobrado = 10;
```

Dado que este es un miembro de la clase, ahora tengo que utilizar el nombre de la clase para acceder a éste, en lugar del nombre de la referencia de la instancia.

Punto del Programador: Los Miembros de Datos Estáticos son Útiles y Peligrosos

Cuando esté recopilando los metadatos necesarios para su proyecto, debe analizar qué cosas pueden ser declaradas estáticas. Cosas como los límites de valores (la edad más avanzada que va a permitir que una persona tenga) pueden ser declarados como de tipo estático. Esto es debido a que puede querer en algún momento cambiar el límite de edad, y no querrá tener que ir uno por uno actualizando todos los objetos declarados en su programa.

Y por supuesto, como dijo el tío de Spiderman, "Un gran poder conlleva una gran responsabilidad". Debe tener cuidado acerca de cómo proporciona acceso a los miembros de datos declarados estáticos. Un simple cambio a un valor estático de un miembro de datos, afectará a todo su sistema. Por lo tanto, estos siempre deben hacerse privados y ser actualizados mediante llamadas a métodos.

4.6.3 Utilizar un método static en una clase

Nosotros también podemos declarar métodos `static`. Hemos realizado este tipo de declaración estática en el método `Main`. Pero también podemos utilizarla cuando diseñamos nuestro sistema. Por ejemplo, podríamos tener un método que determinara si una persona cumple los requisitos requeridos para abrir y disponer de una cuenta bancaria. Los requisitos requeridos podrían estar estipulados en la edad y el ingreso de efectivo. El método devolvería `true` o `false` dependiendo si la persona cumple o no los requisitos requeridos:

```
public bool CuentaConcedida(decimal ingreso, int edad)
{
    if ( (ingreso >= 10000) && (edad >= 18) )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

El método anterior comprueba los valores de la edad y el ingreso en cuenta para una cuenta prospectiva. Se debe tener una edad superior a 17 años y realizar al menos un ingreso en efectivo de 10000 libras para que se conceda abrir una cuenta. El inconveniente que se presenta, es que, por el momento, no podemos llamar a este método hasta que tengamos una instancia de `Cuenta`. Podemos resolver este impedimento declarando el método como `static`:

```
public static bool CuentaConcedida(decimal ingreso, int edad)
{
    if ( (ingreso >= 10000) && (edad >= 18) )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Ahora el método es parte de la clase, y no una instancia de la clase. Por lo que puedo llamar al método utilizando el nombre de la clase:

```
if (Cuenta.CuentaConcedida(25000, 21))
{
      Console.WriteLine("Cuenta concedida");
}
```

Esto es genial porque no he tenido que crear una instancia de la cuenta para realizar la comprobación de si se permite o no crear la cuenta.

Utilizar miembros de datos en métodos estáticos

El método `CuentaConcedida` funciona correctamente, pero por supuesto, he tenido que fijar los valores de la edad y el ingreso. Podría crear un método más flexible que el anterior:

```
public class Cuenta
{
    private decimal ingresoMin = 10000;
    private int edadMin = 18;

    public static bool CuentaConcedida( decimal ingreso, int edad )
    {
       if ( (ingreso >= 10000) && (edad >= 18) )
       {
          return true;
       }
       else
```

```
        {
            return false;
        }
    }
}
```

Este es un diseño mejorado, dado que ahora tengo miembros de la clase que establecen los límites superiores de la edad y el ingreso. Sin embargo, no es una solución válida, ya que la clase anterior no compilará:

```
AdministrarCuenta.cs(19,21): error CS0120: Se requiere una referencia de
objeto para el campo, método o propiedad no estático 'Cuenta.ingresoMin'

AdministrarCuenta.cs(19,43): error CS0120: Se requiere una referencia de
objeto para el campo, método o propiedad no estático 'Cuenta.edadMin'
```

Como de costumbre, el compilador nos está diciendo exactamente lo que está mal; utilizando un lenguaje que nos hace darle vueltas a la cabeza. Lo que el compilador realmente quiere decir es que "*un método estático está haciendo uso de un miembro de la clase que no es estático*".

Si esa indicación no le sirve de ayuda, ¿qué le parece esta?: Los miembros `ingresoMin` y `edadMin` están contenidos dentro de *instancias* de la clase `Cuenta`. Sin embargo, un método estático puede ejecutarse sin necesidad de una instancia (ya que forma parte de la clase). El compilador se muestra disconforme, porque en esta situación el método no tendría ningún miembro con el que interactuar. Podemos corregir esto (y hacer que nuestro programa compile correctamente) declarando también `static` los límites del ingreso y la edad:

```
public class Cuenta
{
    private static decimal ingresoMin;
    private static int edadMin;

    public static bool CuentaConcedida(decimal ingreso, int edad)
    {
        if ( (ingreso >= ingresoMin) && (edad >= edadMin) )
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Código de Ejemplo 33 Cuenta Concedida

Si lo piensa detenidamente, esto tiene un sentido lógico. Los valores límite no deben ser almacenados en cada instancia de clase, ya que queremos que los límites sean los mismos para todas las instancias de la clase, por lo tanto, hacerlos a ellos `static` es lo que nosotros deberíamos haber hecho en primer lugar.

Punto del Programador: Los Miembros de Métodos Estáticos pueden ser utilizados para crear Bibliotecas

A veces en un desarrollo, se necesita proporcionar una biblioteca de métodos para hacer ciertas tareas. El propio sistema C#, cuenta con un gran número de métodos para realizar funciones matemáticas, por ejemplo, `sin` y `cos` (seno y coseno). Tiene sentido hacer este tipo de métodos estáticos, ya que en este tipo de situaciones lo que queremos es acceder al método en sí, no a una instancia de una clase. Una vez más, cuando esté construyendo su sistema, debe pensar en cómo va a disponer tales métodos para su propio uso.

Notas Bancarias: Datos Bancarios Estáticos

La clase de problemas que nosotros podemos resolver utilizando `static` en nuestro banco son:

variable miembro estática: el director gerente quiere que establezcamos la tasa de interés en todas las cuentas de clientes al mismo tiempo. Un único miembro estático de la clase `Cuenta`, proporcionará una variable que puede ser utilizada dentro de todas las instancias de la clase. Pero debido a que sólo hay una copia única de este valor se puede cambiar, y por lo tanto ajustar la tasa de interés para todas las cuentas. Cualquier valor que sea mantenido para todas las clases (los límites de valores son otro ejemplo de esto), es mejor gestionarlo como un valor estático. En el momento en que se vuelve imposible utilizar el modificar `static`, es cuando el director gerente dice "Oh, las cuentas de los niños de cinco años de edad tienen una tasa de interés diferente de las normales". En ese preciso momento, nosotros somos conscientes de que no podemos utilizar `static`, porque necesitamos mantener diferentes valores en algunas de las instancias.

método miembro estático: el director gerente nos comunica que necesitamos un método para determinar si una determinada persona puede o no abrir una cuenta. No puedo hacer que este método sea parte de una instancia de Cuenta porque en el momento en que quisiéramos utilizar el método, esta instancia todavía no se habría generado. Yo debo declarar este método utilizando el modificador `static`, para que pueda ejecutarse sin una instancia.

4.7 La Construcción de Objetos

Hemos visto que nuestros objetos son creados cuando nosotros utilizamos el operador `new`:

```
prueba = new Cuenta();
```

Si observa con detenimiento la línea de código anterior, podrá deducir que tiene mucha similitud con la de una llamada a un método.

Realmente esto es lo que ocurre. Cuando se crea una instancia de una clase, el sistema C# realiza una llamada a un método *constructor* en esa clase. El método constructor es un miembro de la clase, y está ahí para permitir al programador tomar el control y establecer el contenido del nuevo objeto. Una de las reglas de C#, es que cada clase **debe** tener un método constructor al que hay que llamar para crear una nueva instancia.

"Pero espera un minuto", me interrumpes para decir, "Hemos estado creando objetos durante todo este tiempo, y nunca he tenido que proporcionar un método constructor". Esto se debe a que el compilador C# es, para variar, permisivo en esta situación. En lugar de gritarle por no haber proporcionado un método constructor, el compilador mantiene la calma y se encarga de crear discretamente un constructor *predeterminado* por usted y lo utiliza.

Podría pensar que esto es extraño, ya que normalmente el compilador le regaña cuando no hace algo que debería hacer, pero en este caso, simplemente está resolviendo el problema sin informarle de ello. Hay dos maneras de ver esto:

compilador bueno: el compilador está tratando de hacerle la vida más fácil

compilador malo: el compilador sabe que, si crea el método constructor automáticamente, usted sufrirá más tarde las consecuencias, cuando tenga que entender por qué no necesita añadir un método constructor

Como tomarse la acción que realiza en estos casos el compilador, depende de usted.

4.7.1 El Constructor predeterminado

Un método constructor tiene el mismo nombre que la clase que lo contiene, y no devuelve nada. Este es llamado cuando nosotros invocamos al operador `new`. Si no proporciona en el código del programa un método constructor (cosa que hasta ahora no hemos hecho), el compilador crea uno por nosotros.

```
public class Cuenta
{
    public Cuenta()
    {
    }
 }
```

Este es el aspecto del constructor predeterminado. Este es declarado `public`, para que pueda ser accedido desde clases externas que quieran crear instancias de la clase. Este método no acepta parámetros. Si creo mi propio método constructor, el compilador asume que sé lo que estoy haciendo y no añade el código el predeterminado. Este hecho, puede causar problemas, de los que hablaremos más adelante.

4.7.2 Nuestro propio método Constructor

Por diversión, podríamos crear un método constructor que simplemente imprima un mensaje cuando sea llamado:

```
public class Cuenta
{
    public Cuenta()
    {
      Console.WriteLine(" Acabamos de crear una cuenta");
    }
 }
```

Este constructor no es muy constructivo (ho ho), pero al menos nos hace saber que ha sido llamado. Cuando mi programa ejecute la línea:

```
CuentaDeRob = new Cuenta();
```

- el programa imprimirá por pantalla el mensaje:

```
Acabamos de crear una cuenta
```

Tenga en cuenta que hacer esto no es muy lógico, ya que el resultado que se obtendrá serán un montón de impresiones por pantalla que el usuario del programa podría no apreciar, pero este simple código nos permite ver cómo funciona el proceso.

Suministrando la Información al Constructor

Resulta útil poder tomar el control cuando una `Cuenta` es creada, pero sería aún mejor poder asignar valores a los miembros de datos de la cuenta cuando la estoy creando. Como ejemplo,

podría querer al crear la cuenta establecer directamente el nombre, la dirección, y el saldo inicial del titular. En otras palabras, yo quiero hacer lo siguiente:

```
CuentaDeRob = new Cuenta("Rob Miles", "Hull", 0);
```

Este código anterior, crearía una nueva cuenta y establecería la propiedad nombre a `Rob Miles`, la dirección a `Hull` y el saldo inicial a `cero`. Yo puedo hacer esto muy fácilmente, lo único que tengo que hacer es un método constructor que acepte estos parámetros y los utilice para establecer los miembros de la clase:

```
class Cuenta
{
    // miembros de datos privados
    private string nombre;
    private string direccion;
    private decimal saldo;

    // constructor
    public Cuenta(string enNombre, string enDireccion, decimal enSaldo)
    {
       nombre = enNombre;
       direccion = enDireccion;
       saldo = enSaldo;
    }
}
```

Nota del traductor: La traducción de los parámetros del método Cuenta, son abreviaturas del Nombre de entrada (enNombre), Dirección de entrada (enDireccion), y Saldo de entrada (enSaldo).

El constructor toma los valores suministrados en los parámetros, y los utiliza para establecer los miembros de datos de la instancia `Cuenta` que está siendo creada. En este sentido, se comporta exactamente como cualquier otro método.

```
class Banco
{
    public static void Main()
    {
        Cuenta CuentaDeRob;
        CuentaDeRob = new Cuenta("Rob", "Casa de Rob", 1000000);
    }
}
```

Código de Ejemplo 34 Utilizar un Constructor predeterminado

El código anterior crearía una cuenta para Rob con 1000000 libras de saldo. Ojalá se cumpliera.

Tenga en cuenta que añadir un constructor como este tiene una repercusión muy poderosa:

Debe utilizar el nuevo constructor para crear una instancia de una clase, es decir, la única manera ahora en la que puedo crear un objeto `Cuenta`, es suministrando un nombre, una dirección y un saldo inicial. Si intento hacer lo siguiente:

```
CuentaDeRob = new Cuenta();
```

- el compilador dejará de ser agradable conmigo y mostrará el error:

```
PruebaDeCuenta.cs(9,27): error CS1501: Ninguna sobrecarga para el método
'Cuenta' acepta '0' argumentos
```

Lo que el compilador me está indicando es que no hay ningún método constructor en la clase que no tenga parámetros. En otras palabras, el compilador solamente establece un método constructor predeterminado, siempre que el programador no lo proporcione por sí mismo.

Esto puede crear cierta confusión, si hemos hecho uso del constructor predeterminado en nuestro programa, y posteriormente añadimos uno creado por nosotros. El constructor predeterminado ya no será proporcionado por el compilador, y nuestro programa ahora no compilará correctamente. En esta situación, usted tiene que buscar en el código y encontrar todas las llamadas que se produzcan al método constructor predeterminado y actualizarlas, o crear su propio método constructor predeterminado para utilizar estas llamadas. Por supuesto, usted no tiene que hacer esto, porque su diseño del programa seguro que fue tan bueno que no tiene este problema. Como siempre ocurre en mi caso, ejem ejem.

4.7.3 Sobrecarga de Constructores

La sobrecarga es una palabra interesante. En el contexto de la serie de ciencia ficción "Star Trek", es lo que hacen al motor warp en cada uno de los episodios. En el contexto de un programa desarrollado en C# significa:

"Un método que tiene el mismo nombre que otro, pero que cuenta con un conjunto de parámetros diferentes"

El compilador sobrecargará los métodos para usted, determinando a partir de los parámetros suministrados en la llamada a este, qué método de los existentes debe utilizar. En el contexto del constructor de una clase, lo que esto significa es que usted puede proporcionar varias formas diferentes de construir una instancia de una clase. Por ejemplo, muchas (pero no todas) cuentas bancarias, serán creadas con un valor inicial de saldo de cero, es decir, nada en la cuenta. Lo que significa que nos gustaría poder escribir:

```
CuentaDeRob = new Cuenta("Rob", "Hull");
```

He omitido el valor del saldo inicial, dado que quiero utilizar "por defecto" un valor nulo. Al escribir este código, el compilador nada más que busca un método constructor que tenga dos cadenas como parámetros. Uno como este:

```
public Cuenta (string enNombre, string enDireccion)
{
   nombre = enNombre;
   direccion = enDireccion;
   saldo = 0;
}
```

Sobrecargar un nombre de método

En efecto, puede sobrecargar cualquier nombre de método en sus clases. Esto puede serle útil si tiene una acción en particular que puede ser determinada por un número de diferentes elementos de datos, por ejemplo, puede proporcionar varias formas de establecer la fecha de una transacción:

```
EstablecerFecha(int agno, int mes, int dia)

EstablecerFecha(int agno, int fechaJuliana)

EstablecerFecha(string fechaEnMMDDAA )
```

Una llamada del tipo:

```
EstablecerFecha(2005, 7, 23);
```

- correspondería con el método anterior que acepta los tres parámetros enteros, por tanto, el código sería ejecutado por el primer método declarado.

4.7.4 Administración de Métodos Constructores

Si la clase `Cuenta` va a contar con un montón de métodos constructores, esto puede crear confusión al programador:

```
public Cuenta(string enNombre, string enDireccion, decimal enSaldo)
{
   nombre = enNombre;
   direccion = enDireccion;
   saldo = enSaldo;
}

public Cuenta(string enNombre, string enDireccion)
{
   nombre = enNombre;
   direccion = enDireccion;
   saldo = 0;
}

public Cuenta(string enNombre)
{
   nombre = enNombre;
   direccion = "No Suministrada";
   saldo = 0;
}
```

He creado tres métodos constructores para una instancia `Cuenta`. La primera es suministrada con toda la información, la segunda no proporciona ningún saldo y establece el valor a 0. La tercera tampoco recibe la dirección, y establece su valor a "`No Suministrada`".

Para hacerlo he tenido que duplicar el código. Los buenos programadores detestan el código duplicado. Esto es considerado como un "trabajo peligroso extra". Lo realmente preocupante es que es bastante fácil de hacer, tan solamente hace falta utilizar la opción de copiar bloque en el editor de texto, y pegar el fragmento de código seleccionado en cualquier lugar. No debería hacer esto, porque no es recomendable. Si necesita modificar un fragmento de código, tiene que buscar cada copia que haya realizado de ese trozo de código a lo largo de todo el código del programa, y realizar el cambio manualmente.

Esto sucede más a menudo de los que pueda imaginar, incluso aunque su código no tenga ningún error, es posible que tenga que realizar algunas modificaciones si la especificación del programa cambia. Debido a esto, C# proporciona una forma en la que se puede llamar a un constructor desde otro. Considere:

```
public Cuenta (string enNombre, string enDireccion, decimal enSaldo)
{
   nombre = enNombre;
   direccion = enDireccion;
   saldo = enSaldo;
}

public Cuenta (string enNombre, string enDireccion):
  this (enNombre, enDireccion, 0 )
{
}

public Cuenta (string enNombre) :
  this (enNombre, "No Proporcionada", 0 )
{
}
```

La palabra reservada `this` significa "otro constructor en esta clase". Como puede ver en el código del ejemplo anterior, la parte resaltada del código, son llamadas que se realizan al primer constructor. Estas llamadas simplemente pasan los parámetros suministrados, junto con los valores predeterminados que hemos indicado, al constructor "apropiado" para que los trate. Esto significa que la transferencia real de los valores desde el constructor al objeto en sí, solamente sucede en un método, y los otros métodos constructores simplemente hacen llamadas a éste.

La sintaxis de estas llamadas son bastante interesantes, dado que la llamada al constructor tiene lugar antes del cuerpo del método constructor. De hecho, éste se encuentra completamente fuera del bloque. Esto tiene sentido, ya que refleja exactamente lo que está sucediendo. El constructor "`this`" se ejecuta antes de que se ingrese al cuerpo del otro constructor. De hecho, en el código anterior, debido a que la llamada de `this` hace todo el trabajo, el cuerpo del constructor puede estar vacío.

```
class Banco
{
    public static void Main()
    {
        const int MAX_CLIENTES = 100;
        Cuenta[] Cuentas = new Cuenta[MAX_CLIENTES];
        Cuentas[0] = new Cuenta("Rob", "Casa de Rob", 1000000);
        Cuentas[1] = new Cuenta("Jim", "Casa de Jim");
        Cuentas[2] = new Cuenta("Fred");
    }
}
```

Código de Ejemplo 35 Sobrecarga de Constructores

El código del ejemplo anterior demuestra que la sobrecarga del constructor se está realizando correctamente. El código crea una matriz de referencias de Cuenta (denominada `Cuentas`), y a continuación, establece los tres primeros elementos apuntando a instancias de Cuenta. El primer elemento hace referencia a una Cuenta para Rob que cuenta con una dirección completa y 1000000 libras. El segundo elemento (elemento 1) hace referencia a una cuenta para Jim que cuenta también con una dirección y un saldo predeterminado de 0 libras. El tercer elemento (elemento 0) hace referencia a una cuenta para Fred que tiene la dirección predeterminada ("No Proporcionada") y un saldo predeterminado de 0 libras.

Punto del Programador: La construcción de objetos debe ser planeada

La forma en que los objetos van a ser construidos es algo que se debe planear cuidadosamente cuando escribe su programa. Usted debería crear un constructor "maestro" que gestione el método integral de construcción del objeto. A continuación, debería hacer que todos los demás métodos constructores utilicen `this` para acceder al contenido de ese método.

4.7.5 Un constructor no puede fallar

Si ve cualquier película de James Bond normalmente en todas ellas hay un momento, en el que se le dice al agente 007 que el destino del mundo está en sus manos. Fallar no es una opción. Los constructores se encuentran en la misma encrucijada que el agente 007. Los constructores no pueden fallar. Y esto es un problema.

Siempre que hemos escrito métodos en el pasado, nos hemos asegurado de verificar que no pueda tener comportamientos erróneos, de manera que, si hay un error, el método no pueda alterar el estado de nuestro objeto. Por ejemplo, el intento de retirar cantidades negativas de dinero de una cuenta bancaria debe ser rechazado.

La manera en que hemos permitido hasta ahora manipular nuestros objetos está totalmente destinada a asegurarnos que estos no puedan ser quebrantados por las personas que los utilizan. Si intenta hacer algo no razonable en la llamada a un método, éste debe rehusar a realizar esta acción y debe devolver algo que indique que no se ha podido hacer el trabajo.

Así que, nosotros sabemos que cuando creamos un método que modifica los datos de un objeto, tenemos que asegurarnos de que la modificación sea siempre válida. Por ejemplo, nosotros no deberíamos permitir que la siguiente llamada fuese realizada:

```
CuentaDeRob.IngresarEfectivo(1234567890);
```

Habrá un límite superior establecido para la cantidad de dinero en efectivo que se pueda ingresar de una vez, por lo que el método `IngresarEfectivo` deberá rechazar esa operación. Pero, ¿qué podemos hacer para detener el siguiente código?:

```
CuentaDeRob = new Cuenta("Rob", "Hull", 1234567890);
```

Al igual que ocurre con James Bond, a los constructores no se les permite fallar. Ocurra lo que ocurra durante la llamada al constructor, esta será realizada y se creará una nueva instancia.

Esto plantea un problema. Parece que podemos vetar valores que consideremos que no son adecuados en cada punto del programa exceptuando en el más importante, es decir, cuando el objeto es primero creado.

Punto del Programador: El control de errores es un trabajo duro

Esto nos lleva a un tipo de tema recurrente en nuestra búsqueda para convertirnos en grandes programadores. Escribir un código para realizar un trabajo es normalmente muy fácil de hacer. Escribir un código que controle todas las posibles condiciones erróneas de manera eficaz es mucho más complicado. Es un hecho, que en su vida como programador va a pasar más tiempo (o al menos así debería ser) preocupándose por las cosas que fallan, que lo que lo va a hacer por las cosas que funciona correctamente.

Constructores y Excepciones

La única forma de evitar que esto ocurra, es hacer que el constructor lance una excepción si ocurre algún imprevisto. Esto significa que el usuario del constructor debe asegurarse de que se están capturando las excepciones a la hora de crear los objetos, lo que no es nada negativo sino más bien todo lo contrario. La manera más inteligente de hacer esto es haciendo que el constructor llame a los métodos establecidos para cada una de las propiedades que han sido suministradas, y si alguno de ellos devuelve un error, el constructor debe lanzar una excepción en ese punto:

```
public Cuenta(string enNombre, string enDireccion)
{
   if (EstablecerNombre(enNombre ) == false) {
     throw new Exception(enNombre + " es un nombre no válido");
   }
   if (EstablecerDireccion(enDireccion) == false) {
     throw new Exception(enDireccion + " es una dirección no válida");
   }
}
```

Si se intenta crear una cuenta con un nombre no razonable (no válido), se lanzará una excepción, tal y como nosotros queremos que ocurra. El único problema aquí es que si la dirección tampoco

es razonable (no válida), el usuario no lo sabrá hasta que haya corregido el nombre y haya llamado al constructor de nuevo.

Odio cuanto estoy utilizando un programa y esto ocurre. Normalmente me ocurre a la hora de rellenar un formulario en la web. Si he introducido mi nombre de manera incorrecta, se me informa de ello. Cuando lo corrijo, entonces se me informa de otro error que he cometido al introducir la dirección. Lo que yo quiero es poder indicar todos los campos no válidos de una sola vez. Esto puede hacerse, a expensas de un código algo más complejo:

```
public Cuenta(string enNombre, string enDireccion, decimal enSaldo)
{
   string mensajeError = "";

   if (EstablecerSaldo(enSaldo) == false)
   {
     mensajeError = mensajeError + enSaldo + " saldo no válido";
   }

   if (EstablecerNombre(enNombre) == false)
   {
     mensajeError = mensajeError + enNombre + " nombre no válido";
   }

   if (EstablecerDireccion(enDireccion) == false)
   {
     mensajeError = mensajeError + enDireccion + " es una dirección no
                                                  válida";
   }

   if (mensajeError != "")
   {
     throw new Exception("Creación de la cuenta fallida " + mensajeError);
   }
}
```

Código de Ejemplo 36 Constructor fallido

Esta versión del constructor ensambla un mensaje de error que describe todos los campos que se han rellenado erróneamente en la cuenta. Cada nuevo campo que se rellena de manera no válida, es añadido al mensaje que se muestra en pantalla cuando la excepción es lanzada. A continuación, el programa devuelve el control del programa al método desde el que se realizó la llamada.

Punto del Programador: Considere las cuestiones internacionales

El código anterior ensambla un mensaje de texto y lo envía al usuario cuando algo inesperado ocurre. Esto es algo positivo. No obstante, si escribe el programa como se muestra arriba, esto podría causar un problema cuando ese mismo código se utilice en una sucursal francesa del banco. Durante el proceso de especificación, necesita conocer si el código necesita ser desarrollado para múltiples idiomas. Si esto es así, necesitará administrar los datos almacenados y seleccionar los mensajes apropiados. Afortunadamente, existen algunas bibliotecas en C# que han sido diseñadas para hacer este trabajo más fácilmente.

NOTAS BANCARIAS: CONSTRUYENDO UNA CUENTA

Los problemas que giran en torno al constructor de una clase no son directamente relevantes para la especificación de la cuenta bancaria como tal, ya que estos realmente se relacionan con la forma en que la especificación es implementada, y no lo que el sistema en realidad hace.

Dicho esto; si el gerente dice algo como: "El cliente rellena un formulario, introduce su nombre y dirección, y estos datos se utilizan para crear la nueva cuenta", esto le da una buena idea de qué parámetros se deben suministrar al constructor.

4.8 De objeto a componente

Considero que, en determinadas fases del desarrollo de un software, uno siempre pasa por el proceso de "dar un paso atrás" desde el problema y pensar en niveles cada vez más altos. La gente pija lo llama "abstracción". Este es el progreso que nosotros hemos realizado hasta ahora:

- representar valores identificados por nombres (variables)
- crear acciones que trabajan sobre las variables (sentencias y bloques)
- establecer comportamientos dentro de bloques de código a los que podemos asignarles nombres identificativos. Podemos reutilizar estos comportamientos y también usarlos en el proceso de diseño (métodos)
- crear elementos que contienen variables miembros en calidad de propiedades, y métodos miembros en calidad de acciones (objetos)

En lugar de perder mucho tiempo al comenzar un proyecto, preocupándonos solamente de cómo vamos a representar una cuenta y precisar lo que esta debe hacer, nosotros solamente debemos decirnos “Necesitamos una cuenta", y luego pasar a otras cosas. Más tarde regresaremos al punto inicial y revisaremos el problema con un mayor nivel de detalle, desde el punto de vista de lo que la clase Cuenta necesitamos que haga.

El siguiente paso a considerar es cómo dar un paso más hacia atrás y expresar una solución utilizando *componentes* e *interfaces*. En esta sección, descubrirá las diferencias entre un objeto y un componente, y cómo utilizar estos últimos para diseñar un sistema.

4.8.1 Componentes y Hardware

Antes de empezar a pensar en las cosas desde el punto de vista del software, es apropiado considerar las cosas desde el punto de vista del hardware. Debe estar familiarizado con la forma en que una típica computadora de hogar, tiene componentes que no están integrados en la placa principal. Por ejemplo, el adaptador de gráficos. Esto es ideal; ya que de esta forma se puede adquirir un nuevo adaptador gráfico en cualquier momento, y conectarlo a la máquina para mejorar el rendimiento del equipo.
Para que esto funcione correctamente, los diseñadores que fabrican placas base y los diseñadores que fabrican adaptadores gráficos, se han unido y han llegado a un acuerdo para crear una *interfaz* de comunicación entre estos dos dispositivos. Esta interfaz toma la forma de un extenso documento que describe exactamente cómo interactúan los dos componentes, por ejemplo, cuales señales son de entrada, cuales señales son de salida, etc. Cualquier placa principal que contenga un zócalo construido según el estándar puede admitir una tarjeta gráfica.

Así pues, desde el punto de vista del hardware, los componentes son posibles porque nosotros hemos creado *interfaces* que describen con precisión como se interrelacionan entre sí.

Los componentes de software son **exactamente lo mismo**.

¿Por qué necesitamos componentes de software?

Por el momento, podría no apreciar la necesidad de utilizar componentes de software. Cuando estamos creando un sistema, trabajamos en lo que cada una de las partes tiene que hacer, y luego, creamos esas partes. En esta etapa todavía no es obvio comprender la razón de por qué los componentes son necesarios.

Pues un sistema diseñado sin componentes es exactamente como una computadora con un adaptador de gráficos integrado en la placa base. No es posible mejorar el adaptador de gráficos porque este está "integrado" dentro del sistema.

Sin embargo, desafortunadamente, en nuestro sistema bancario podríamos tener la necesidad de crear diferentes formas de clase de cuenta bancaria. Por ejemplo, es posible que se nos pueda pedir crear una clase "`CuentaInfantil`", que solamente permita que el titular de la cuenta retire un máximo de diez libras en cada operación. Esto puede suceder incluso después de que nosotros hayamos instalado el sistema y ya esté siendo utilizado.

Si todo ha sido integrado, esto será imposible de hacer. Al describir los objetos en términos de sus interfaces, sin embargo, nosotros podemos utilizar cualquier elemento que se comporte como una `Cuenta` en esta situación.

4.8.2 Componentes e Interfaces

Una aclaración, que yo debería hacer aquí es que nosotros **no** estamos hablando de la interfaz de usuario de nuestro programa. La interfaz de usuario es el medio con que el usuario puede comunicarse con una computadora, y comprende todos los puntos de contacto entre el usuario y el equipo. Estas generalmente suelen ser de tipo texto (el usuario introduce comandos y espera la respuesta por parte de la máquina) o gráficas (el usuario hace clic en los "botones" que se muestran en la pantalla utilizando el ratón).

Una *interfaz*, por otro lado, solo especifica cómo un componente de software podría ser utilizado por otro componente de software. Por favor, que no se le ocurra responder a una pregunta de examen destinada a explicar el mecanismo de una interfaz en C#, con una larga descripción de cómo funcionan las ventanas y los botones. Esto le hará obtener cero puntos.

Interfaces y Diseño

Así que, en lugar de empezar diseñando las clases, nosotros deberíamos hacerlo pensando en describir sus interfaces, es decir, lo que ellas tienen que hacer. En C#, nosotros expresamos esta información en un elemento llamado *interface*. Una interfaz es simplemente un conjunto de definiciones de métodos agrupados lógicamente por estar relacionados entre sí para llevar a cabo una tarea o rol en particular.

Nuestro primer paso en una interfaz de cuenta bancaria podría ser el siguiente:

```
public interface ICuenta
{
   void IngresarEfectivo(decimal cantidad);
   bool RetirarEfectivo(decimal cantidad);
   decimal ObtenerSaldo();
}
```

Este código indica que la interfaz `ICuenta` está compuesta de tres métodos, uno para ingresar dinero en efectivo; otro para retirarlo y un tercero que devuelve el saldo en la cuenta. Desde el punto de vista de la gestión del saldo, esto es todo lo que necesitamos. Tenga en cuenta que, a nivel de interfaz, no estoy indicando cómo debe implementarse o construirse, solamente estoy indicando lo que debe hacer. Una interfaz es a nivel de código fuente como una clase, y se compila exactamente de la misma manera. Como hemos indicado, la interfaz establece una serie de métodos relacionados con una tarea o rol en particular, en este caso, lo que una clase debe hacer para ser considerada una cuenta bancaria.

Punto del Programador: Los nombres de las interfaces comienzan por la letra I

Hemos visto que existen una serie de recomendaciones o reglas a seguir denominadas convenciones de código, las cuales establecen por ejemplo un estándar para otorgar nombres a las variables. Otra regla de esta convención es que el nombre de una interfaz debe comenzar con la letra `I`. Si no sigue esta regla, su programa compilará correctamente, ya que el compilador no tiene ninguna opinión sobre como nombramos a las variables, pero usted puede tener problemas para dormir por las noches. Y, además, con toda lógica.

4.8.3 Implementar una Interfaz en C#

Las interfaces se vuelven interesantes cuando creamos una clase que las *implementa*. Implementar una interfaz es similar a establecer un contrato entre el proveedor de recursos y el consumidor. Si una clase implementa una interfaz, está indicando que, para cada método descrito en la interfaz, cuenta con una implementación correspondiente.

Son, por tanto, las clases que implementen estas interfaces las que describen la lógica del comportamiento de los métodos.

En el caso de la cuenta bancaria, voy a crear una clase que implemente la interfaz, para que ésta pueda ser considerada como un componente de la cuenta, independientemente de lo que realmente es:

```
public class CuentaCliente : ICuenta
{
    private decimal saldo = 0;

    public bool RetirarEfectivo(decimal cantidad)
    {
       if (saldo < cantidad)
       {
          return false;
       }
       saldo = saldo - cantidad;
       return true;
    }

    public void IngresarEfectivo(decimal cantidad)
    {
       saldo = saldo + cantidad;
    }
```

```
        public decimal ObtenerSaldo()
        {
            return saldo;
        }
    }
```

El código no difiere demasiado del de la clase previa vista anteriormente. La única diferencia se encuentra en la línea superior:

```
public class CuentaCliente : ICuenta
{
    ...
```

La línea resaltada en el código anterior, es en donde el programador le comunica al compilador que esa clase implementa la interfaz `ICuenta`. Esto significa que la clase contiene versiones específicas de todos los métodos descritos en la interfaz. Si la clase no contiene un método que la interfaz necesita, se producirá un error de compilación:

```
error CS0535: 'AdministrarCuenta.CuentaCliente' no implementa el miembro
de interfaz 'AdministrarCuenta.ICuenta.IngresarEfectivo(decimal)'
```

En este caso, olvidé el método `IngresarEfectivo`, y el compilador se quejó en consecuencia.

4.8.4 Referencias a interfaces

Una vez que hemos compilado la clase `CuentaCliente`, ahora tenemos un elemento que puede ser considerado de dos maneras:

- como una **`CuentaCliente`** (porque eso es lo que esta es)
- como una **`ICuenta`** (porque eso es lo que esta puede hacer)

La gente hace esto todo el tiempo. Usted puede pensar en mí de un montón de maneras distintas, he aquí dos de ellas:

- Rob Miles, la persona (porque eso es lo que yo soy)
- Un conferenciante universitario (porque eso es lo que yo puedo hacer)

Si piensa en mí como un conferenciante universitario, estaría utilizando una interfaz que contiene métodos como `DarPonencia`. Y podría utilizar los mismos métodos con cualquier otro conferenciante (es decir, persona que implementa esa interfaz). Desde el punto de vista de la universidad, que tiene que gestionar una gran cantidad de conferenciantes intercambiables, es mucho más útil pensar en mí como un conferenciante universitario, que, como Rob Miles, la persona.

Así pues, con las interfaces nos estamos alejando de considerar las clases en términos de lo que son, y empezamos a pensar en ellas en términos de lo que pueden hacer. En el caso de nuestro banco, esto significa que queremos tratar con los objetos en términos de `ICuenta`, (el conjunto de capacidades de la cuenta) en lugar de como `CuentaCliente` (una clase de cuenta particular).

En términos de C#, esto significa que necesitamos ser capaces de crear variables de referencia, que se refieran a los objetos en términos de las interfaces que implementan, en lugar de en el tipo particular que son. Esto es bastante fácil de conseguir:

```
ICuenta cuenta = new CuentaCliente();
cuenta.IngresarEfectivo(50);
Console.WriteLine("Saldo: " + cuenta.ObtenerSaldo());
```

Código de Ejemplo 37 Interfaz simple

La variable `cuenta` está autorizada para apuntar a los objetos que implementan la interfaz `ICuenta`. El compilador realizará la comprobación para asegurarse de que `CuentaCliente` hace esto, y si lo hace, la compilación será realizada correctamente.

Tenga en cuenta que nunca habrá una instancia de la interfaz `ICuenta`. Esta es simplemente una forma en las que nosotros podemos referirnos a alguna cosa que tiene esa capacidad (es decir, contiene los métodos requeridos).

Esto es lo mismo que en la vida real. No existe una cosa física como tal que sea un "conferenciante", solamente una gran cantidad de personas a las que se puede hacer referencia como que tienen esa capacidad o rol en particular.

4.8.5 Utilizar interfaces

Ahora que tenemos nuestro sistema diseñado con interfaces es mucho más fácil extenderlo. Puedo crear una clase `CuentaInfantil` que implemente la interfaz `ICuenta`. Esta implementa todos los métodos requeridos, pero se comporta de manera ligeramente diferente porque nosotros queremos que todas las retiradas de efectivo superiores a diez libras fallen:

```
public class CuentaInfantil : ICuenta
{
    private decimal saldo = 0;

    public bool RetirarEfectivo(decimal cantidad)
    {
       if ( saldo > 10)
       {
          return false;
       }
       if (saldo < cantidad)
       {
          return false;
       }
       saldo = saldo - cantidad;
       return true;
    }

    public void IngresarEfectivo(decimal cantidad)
    {
       saldo = saldo + cantidad;
    }

    public decimal ObtenerSaldo()
    {
       return saldo;
    }
}
```

Lo bueno de esto es que, como es un componente, nosotros no tenemos que modificar todas las clases que la utilicen. Cuando creamos los objetos de la cuenta, solamente tenemos que preguntar si se requiere una cuenta estándar o una cuenta infantil. El resto del sistema, puede entonces tomar este objeto y utilizarlo sin preocuparse realmente de lo que es. Por supuesto, tendremos que crear algunas pruebas destinadas especialmente para ello, para asegurarnos de que los retiros de más de diez libras fallan, pero utilizar el nuevo tipo de cuenta en nuestro Sistema actual es muy fácil.

Esto significa que podemos crear componentes que son utilizados de la misma manera, pero que tienen comportamientos diferentes apropiados para el tipo de elemento que están representando.

```
class Banco
{
    const int MAX_CLIENTES = 100;

    public static void Main()
    {
        ICuenta[] cuentas = new IAccount[MAX_CLIENTES];

        cuentas[0] = new CuentaCliente();
        cuentas[0].IngresarEfectivo(50);
        Console.WriteLine("Saldo: " + cuentas[0].ObtenerSaldo());

        cuentas[1] = new CuentaInfantil();
        cuentas[1].IngresarEfectivo(20);
        Console.WriteLine("Saldo: " + cuentas[1].ObtenerSaldo());

        if (cuentas[0].RetirarEfectivo(20))
        {
            Console.WriteLine("Retirada de efectivo correcta");
        }

        if (cuentas[1].RetirarEfectivo(20))
        {
            Console.WriteLine("Retirada de efectivo correcta");
        }
        Console.ReadKey();
    }
}
```

Código de Ejemplo 38 Utilizar Componentes

El código anterior muestra cómo se utilizaría realmente la interfaz. La matriz de cuentas puede contener referencias a cualquier objeto que implemente la interfaz `ICuenta`. Esto incluye los objetos tanto de la `CuentaCliente` como de la `CuentaInfantil`. El primer elemento de la matriz (elemento 0) es asignado a un objeto `CuentaCliente`, y el segundo elemento (elemento 1) es asignado a un objeto `CuentaInfantil`. Cuando los métodos son invocados sobre estos objetos, se utiliza el que se encuentra en el tipo particular. Esto significa que la llamada final al método `RetirarFondos` fallará porque, aunque la cuenta infantil tenga suficiente dinero en el banco, no se les permite retirar más de 10 libras.

4.8.6 Implementar Múltiples Interfaces

Un componente puede implementar tantas interfaces como sean requeridas. La interfaz `ICuenta` me permite considerar un componente puramente en términos de su capacidad de comportarse como una cuenta bancaria. Sin embargo, es posible que desee considerar un componente en una

variedad de formas distintas. Por ejemplo, el banco querrá que la cuenta se pueda imprimir en papel.

Puede pensar que todo lo que yo tengo que hacer es añadir un método de impresión a la interfaz `ICuenta`. Esto sería razonable si todo lo que se quisiera imprimir fuesen cuentas bancarias. Sin embargo, habrá una gran cantidad de cosas que necesiten ser impresas, por ejemplo, cartas de aviso, ofertas especiales y similares. Cada uno de estos elementos se implementará en términos de un componente que proporciona una interfaz particular (por ejemplo, `IAviso`, `IOfertaEspecial`). No quiero tener que proporcionar un método de impresión en cada uno de estos elementos, lo que realmente quiero es una forma en que pueda considerar un objeto en términos de su capacidad para imprimir.

Realmente, es muy fácil conseguir esto. Construyo la interfaz:

```
public interface IImprimirEnPapel
{
   void Imprimir();
}
```

Ahora cualquier elemento que implemente la interfaz `IImprimirEnPapel`, contendrá el método `Imprimir` y puede ser considerado en términos de su capacidad para imprimir.

Una clase puede implementar tantas interfaces como sea necesario. Cada interfaz es una nueva forma en la que nos podemos referir y acceder a ella.

```
public class CuentaInfantil : ICuenta, IImprimirEnPapel
{
 …
```

Esto significa que una instancia `CuentaInfantil` se comporta como una cuenta, y también contiene un método `Imprimir` que puede ser utilizado para imprimir.

4.8.7 Diseño con Interfaces

Si aplica la técnica de "abstracción" de manera apropiada, debería terminar con un proceso de creación de Sistema en consonancia con las siguientes líneas:

- recopilar tantos *metadatos* como pueda sobre el problema; lo que es importante para el cliente, qué valores necesitan ser representados y manipulados y el rango de esos valores.
- identificar las clases que tenemos que crear para representar los componentes en el problema.
- identificar las acciones (métodos) y los valores (propiedades) que los componentes deben proporcionar.
- incluir éstas dentro de interfaces para cada uno de los componentes.

- decidir cómo estos valores y acciones van a ser testeados.
- implementar los componentes y testearlos sobre la marcha.

Puede y debe considerar estos procesos en papel, antes de escribir siquiera una línea de código. También hay herramientas gráficas que puede utilizar para dibujar diagramas formales para representar esta información. El campo de la Ingeniería del Software está basado completamente en este proceso.

También debería haber observado que las interfaces son elementos ideales para exponer las pruebas de testeo. Si cuenta con un conjunto de comportamientos fundamentales que todas las cuentas bancarias deben tener (por ejemplo, la acción de ingresar efectivo siempre debe hacer que el saldo de la cuenta incremente), entonces puede escribir dichas pruebas unidas a la interfaz `ICuenta`, y utilizarlas para comprobar cualquiera de los componentes bancarios creados.

Punto del Programador: Las interfaces son solo promesas

Una interfaz es más una promesa que un contrato vinculante. Tan solo porque una clase tenga un método llamado **`IngresarEfectivo`**, no significa que éste ingrese dinero en la cuenta; solamente significa que existe un método con ese nombre dentro de la clase. No existe nada en C# que permita imponer un comportamiento particular en un método; por lo que esto viene en función de cuánto confíe en el programador que hizo la clase que se vaya a utilizar o esté utilizando, y cómo de coherentes sean sus pruebas. De hecho, a veces lo usamos fructíferamente cuando desarrollamos un programa, para crear componentes "ficticios" que implementan la interfaz pero que no tienen el comportamiento como tal.

El mecanismo de interfaz nos brinda una gran flexibilidad cuando creamos nuestros componentes y los unimos. Esto significa que una vez que hayamos descubierto lo que nuestra clase de cuenta bancaria necesita mantener para nosotros, entonces podemos a continuación considerar qué es lo que le pediremos a las cuentas que hagan. Este es el auténtico matiz de la especificación. Una vez que hemos establecido una interfaz para un componente, podemos pensar simplemente en términos de lo que el componente debe hacer, y no precisamente cómo lo hace.

Por ejemplo, el director gerente nos ha dicho que cada cuenta bancaria debe tener un número identificativo de cuenta. Este es un valor muy importante, ya que será fijado desde el inicio, y no podrá ser cambiado durante la vida de la cuenta. No existirán dos cuentas que utilicen el mismo número.

Desde el punto de vista del diseño de la interfaz esto significa que el número de cuenta se establecerá cuando se cree la cuenta, y la clase de la cuenta proporcionará un método para que obtengamos el valor (pero no habrá un método como tal para establecer el número de cuenta).

No nos importa lo que el método `ObtenerNumeroCuenta` realmente haga a nivel interno, siempre y cuando siempre devuelva el valor de una cuenta en particular. Así que, este requisito termina expresándose en la interfaz que implementa la clase de cuenta de la siguiente manera:

```
interface ICuenta
{
   int ObtenerNumeroCuenta();
}
```

Este método devuelve el número entero que pertenece al número de cuenta para esta instancia. Al colocarlo en la interfaz podemos decir que la cuenta debe entregar este valor, pero en realidad no hemos descrito cómo debe hacer esto. El diseño de las interfaces en un sistema es simplemente esto. Las interfaces manifiestan que necesitamos poder utilizar una serie comportamientos, pero no tienen porque necesariamente indicar cómo van a lograr realizar ese trabajo. Yo he añadido un mayor número de observaciones para proporcionar más detalles sobre lo que hace el método.

La necesidad de tener números identificativos de cuenta, que además necesitan ser únicos en el mundo, ha dado como resultado la implementación y creación de un conjunto de métodos en las bibliotecas C# denominado *Identificador Único Global* (*Globally Unique Identifiers)* o *GUIDs*. Estos son elementos de datos que se crean en función de la fecha, hora y cierta información obtenida del hardware de la computadora host. Cada GUID es único en el mundo. Nosotros podríamos utilizarlos en nuestro constructor de `Cuenta` creando un GUID que permita que cada cuenta tenga un número identificativo único.

4.9 Herencia

La herencia es otro concepto en donde podemos hacer efectiva la pereza creativa. Esta es una forma en la que podemos coger comportamientos implementados en las clases, y modificar las partes del código que necesitemos para hacer otros nuevos. A este respecto, puede considerarse como un mecanismo de *reutilización de código*. Este mecanismo también puede ser utilizado en la etapa de diseño de un programa, si cuenta con un conjunto de objetos relacionados entre sí que desee crear.

La herencia permite que una clase aproveche las características y comportamientos de su clase padre. Una clase que implemente una interfaz debe implementar los comportamientos de la interfaz especificados en la definición de interfaz. Si una clase desciende de otra clase particular, significa que *hereda* el conjunto de comportamientos de la clase padre. En resumen:

Interfaz: "Puedo hacer estas cosas porque yo he indicado que puedo hacerlas"
Herencia: "Puedo hacer estas cosas porque mi padre puede hacerlas"

4.9.1 Extender una clase padre

Vamos a ver un ejemplo de uso de herencia en nuestro proyecto de cuenta bancaria. Nosotros hemos mencionado que una `CuentaInfantil` debe comportarse como una `CuentaCliente` excepto en lo que respecta al método de retirada de efectivo. Las cuentas de cliente estándar pueden retirar todo el efectivo que necesiten. Las cuentas Infantiles solamente están autorizadas para retirar 10 libras en cada operación.

Hemos resuelto este problema desde el punto de vista del diseño mediante el uso de interfaces. Separando lo que hace el trabajo de la descripción del trabajo (que es lo que una interfaz permite hacer) podemos hacer que todo el sistema bancario piense en términos de una `ICuenta`, y posteriormente conectar cuentas con diferentes comportamientos según sea necesario. Podemos crear nuevos tipos de cuentas incluso después de que el sistema haya sido desplegado. Estas pueden ser introducidas y trabajar junto a las demás porque se comportan correctamente (es decir, implementan la interfaz).

Pero esto hace que las cosas se vuelvan un poco tediosas a la hora de escribir las líneas de código que requerimos en el programa. Necesitamos crear una clase `CuentaInfantil` que contenga un montón de código duplicado de la clase `CuentaCliente`. Usted pensará "Sin problema. Puedo copiar el bloque de código que necesito utilizando el editor, y a continuación, pegarlo en esa parte del programa". Pero:

Punto del Programador: La copia de bloques de código es perjudicial

Yo todavía cometo errores cuando programo. Podría pensar que después de tantos años que llevo dedicado a este trabajo, lo hago todo perfecto de una sola vez. Está equivocado. Y muchos de los errores que cometo son causados por un uso incorrecto de la copia de bloques de código. A veces al estar desarrollando un programa, me doy cuenta que necesito un código similar, pero no exactamente el mismo código que dispongo en otra parte del programa. Así que, simplemente copio ese bloque de código que tengo, y lo pego en donde lo necesito. A continuación, yo realizo los cambios de la parte del código que necesito variar y descubro que mi programa no funciona correctamente.

Intente no hacer esto. Un buen programador escribe cada fragmento de código una vez, y solo una vez. Si necesita utilizar ese trozo de código en otra parte del programa, cree un método para ello.

Lo que nosotros realmente queremos hacer es tomar todos los comportamientos disponibles en la `CuentaCliente`, y luego simplemente realizar los cambios oportunos en el único método que necesitamos que tenga un comportamiento diferente. Resulta que podemos hacer esto en C#

utilizando la herencia. Cuando creo la clase `CuentaInfantil`, yo puedo indicarle al compilador que ésta se basa en la `CuentaCliente`:

```
public class CuentaInfantil : CuentaCliente, IImprimirEnPapel
{
}
```

La clave aquí es la parte resaltada tras el nombre de la clase. Yo establezco en la definición el nombre de la clase de la que `CuentaInfantil` extiende. Esto significa que todo lo que clase `CuentaCliente` puede hacer, la clase `CuentaInfantil` puede también hacerlo.

A continuación, puedo escribir un código como este:

```
CuentaInfantil b = new CuentaInfantil();
b.IngresarEfectivo(50);
```

Este código funciona porque, aunque `CuentaInfantil` no contiene un método `IngresarEfectivo`, su clase padre se encarga de realizar el trabajo. Lo que significa que el método `IngresarEfectivo` de la clase `CuentaCliente` es utilizado en este punto.

Por tanto, debe tener en cuenta que las instancias de la clase `CuentaInfantil` incoporan las capacidades que recogen de su clase padre. De hecho, en este momento, la clase `CuentaInfantil` no tiene ningún comportamiento propio; así que ésta hereda todo de su padre.

4.9.2 Sobrescritura de métodos

Ahora sabemos que podemos crear una nueva clase basada en una existente. Lo siguiente que necesitamos saber hacer es cambiar el comportamiento del método que nos interesa. Queremos reemplazar el método `RetirarEfectivo` por otro nuevo. Esto es denominado *sobreescribir* un método. En la clase `CuentaInfantil`, puedo hacerlo de la siguiente manera:

```
public class CuentaInfantil : CuentaCliente,ICuenta
{

    public override bool RetirarEfectivo ( decimal cantidad )
    {
       if ( cantidad > 10)
       {
          return false;
       }
       if ( saldo < cantidad)
       {
          return false;
       }
```

```
            saldo = saldo - cantidad;
            return true;
        }
    }
```

El modificador *override* significa "utiliza esta versión del método en preferencia a la del padre". Esto significa que en un código como:

```
CuentaInfantil b = new CuentaInfantil();
b.IngresarEfectivo (50);
b.RetirarEfectivo(5);
```

La llamada de `IngresarEfectivo` utilizará el método del padre (dado que no ha sido anulado), pero la llamada a `RetirarEfectivo` utilizará el método de la `CuentaInfantil`.

Métodos virtuales

Realmente, hay otra cosa que debemos hacer para que la sobreescritura funcione. El compilador de C# necesita saber si un método va a ser anulado en una clase derivada. Esto se debe a que el compilador debe llamar a un método de reemplazo de una manera ligeramente diferente a como lo hace con uno "normal". En otras palabras, el código anterior no compilará correctamente, ya que el compilador no ha sido informado de que `RetirarEfectivo` puede ser anulado/invalidado en clases derivadas.

Para que la sobreescritura funcione correctamente, tengo que cambiar la declaración del método que hice en la clase `CuentaCliente`.

```
public class CuentaCliente : ICuenta
{
    private decimal saldo = 0;

    public bool RetirarEfectivo(decimal cantidad)
    {
        if (saldo < cantidad)
        {
            return false;
        }
        saldo = saldo - cantidad;
        return true;
    }
}
```

La palabra reservada `virtual` significa “Es posible que yo quiera crear otra versión de este método en una clase hija”. Usted no tiene realmente que proporcionar el método de reemplazo, pero si no tiene la palabra designada, definitivamente no podrá hacerlo.

Esto hace que `override` y `virtual` constituyan una especie de dupla inseparable. Puede utilizar `virtual` para marcar un método que puede ser anulado o invalidado en una clase derivada, y `override` para realmente proporcionar un método de reemplazo.

Protección de datos en una estructura jerárquica de clases

Resulta que el código anterior aún no funcionará. Esto es porque el valor del saldo en la clase `CuentaCliente` está declarado con el modificador de acceso de miembro `private`. Nosotros lo hicimos privado para que los métodos de otras clases no pudieran acceder al valor y cambiarlo directamente.

Sin embargo, esta protección es demasiado estricta, ya que impide a la clase `CuentaInfantil` cambiar el valor. Para sortear este problema, C# dispone de un nivel de acceso ligeramente menos restrictivo denominado `protected`. Este hace que el miembro sea visible para las clases que extiendan del padre. En otras palabras, los métodos de la clase `CuentaInfantil` pueden ver y utilizar un miembro protegido porque se encuentran en la misma clase jerárquica que la clase que contiene el miembro.

Una jerarquía de clase es parecida a un árbol genealógico. Cada clase tiene un padre y puede realizar todas las acciones o tareas que el padre puede hacer. Esta también tiene acceso a todos los miembros protegidos de las clases superiores a ella.

```
public class CuentaCliente : ICuenta
{
    protected decimal saldo = 0;

    ...
}
```

No estoy del todo satisfecho con este cambio, el saldo de la cuenta es muy importante para mí, y preferiría que nada fuera de la clase `CuentaCliente` pudiera verlo. Sin embargo, por ahora al haber realizado este cambio el programa funciona. Más adelante, veremos mejores maneras de manejar esta situación.

```
ICuenta[] cuentas = new ICuenta[MAX_CLIENTES];

cuentas[0] = new CuenteCliente();
cuentas[0].IngresarEfectivo(50);
Console.WriteLine("Saldo: " + cuentas[0].ObtenerSaldo());

cuentas[1] = new CuentaInfantil();
cuentas[1].IngresarEfectivo(20);
Console.WriteLine("Saldo: " + cuentas[1].ObtenerSaldo());

if (cuentas[0].RetirarEfectivo(20))
{
    Console.WriteLine("Retirada de efectivo correcta");
}

if (cuentas[1].RetirarEfectivo(20))
{
    Console.WriteLine("Retirada de efectivo correcta");
}
```

Código de Ejemplo 39 Utilizar Herencia

El código anterior muestra cómo podemos crear y utilizar diferentes tipos de cuentas, estando la `CuentaInfantil` basada en el tipo de `CuentaCliente` padre. Si en este punto está teniendo una sensación de *déjà vu*, es porque este es exactamente el mismo código que vimos en el ejemplo anterior. El código que usa estos objetos funcionará exactamente de la misma manera que siempre, exceptuando que los objetos en sí trabajan de una manera ligeramente diferente, haciendo que el programa sea más pequeño (lo que no nos importa particularmente), y mucho más fácil de depurar (porque un error en cualquiera de los métodos compartidos sólo tiene que ser corregido una vez).

Notas Bancarias: Reemplazar por Diversión y Beneficio

La capacidad de proporcionar un método de reemplazo es muy poderosa. Esto significa que nosotros podemos hacer clases más generales (por ejemplo, `CuentaCliente`), y personalizarlas para hacerlas más específicas (por ejemplo, `CuentaInfantil`). Por supuesto, este proceso debe planificarse y gestionarse en la etapa de diseño. Esto requiere que se recopilen más *metadatos* del cliente, y se analicen para decidir qué partes del comportamiento se necesitan cambiar durante la vida del proyecto. Nosotros habríamos creado el método `RetirarEfectivo` utilizando el modificador `virtual`, si el director gerente hubiera dicho “Nos gustaría poder personalizar la forma en que algunas cuentas pueden retirar efectivo”. Y lo habríamos escrito en la especificación.

4.9.3 Utilizar el método base

Recuerde que los programadores son esencialmente personas perezosas que intentan escribir el código tan solamente una vez para un problema planteado. ¡Vaya!, parece que estamos rompiendo nuestras propias reglas aquí, dado que el método `RetirarFondos` de la clase `CuentaInfantil` contiene todo el código del método de la clase padre.

Ya hemos indicado anteriormente que esto no es de nuestro agrado, ya que esto significa que el valor del saldo tiene que estar más expuesto de lo que nos gustaría. Afortunadamente, los diseñadores de C# han pensado en esto, y han proporcionado una manera en la que usted puede llamar al método *base* desde uno que lo anule o lo invalide.

La palabra `base` en este contexto significa "una referencia al elemento que ha sido anulado o invalidado". Puedo utilizarla para hacer el método `RetirarEfectivo` en mi `CuentaInfantil` mucho más simple:

```
public class CuentaInfantil : CuentaCliente, ICuenta
{

    public override bool RetirarEfectivo(decimal cantidad)
    {
       if (cantidad > 10)
       {
          return false;
       }
       return base.RetirarEfectivo(cantidad);
    }
}
```

Código de Ejemplo 40 Sobreescribir el metodo base

La última línea del método `RetirarEfectivo`, realiza una llamada al método original `RetirarEfectivo` en la clase padre, es decir, la que el método anula o invalida.

Es importante que entienda lo que estoy haciendo aquí, y por qué lo estoy haciendo:

- No quiero tener escrito el mismo código dos veces.
- No quiero que el valor del saldo sea visible fuera de la clase `CuentaCliente`.

Al utilizar la palabra `base` para llamar al método anulado o invalidado, resuelvo ambos problemas de una manera bastante eficiente. Dado que la llamada al método devuelve un resultado `bool`, puedo simplemente enviar lo que éste me devuelva. Al hacer este cambio, puedo volver a establecer el saldo con el modificador de acceso `private` en la `CuentaCliente` porque éste no se cambia fuera de ella.

Tenga en cuenta que, en este punto, se producen otras repercusiones indirectas beneficiosas para nosotros. Si necesitara corregir un error producido en el comportamiento del método `RetirarFondos`, solamente tendré que corregirlo una vez, en la clase de nivel superior, y este será subsanado para todas las clases que devuelven la llamada al mismo.

4.9.4 Crear un Método de Reemplazo

Esta parte es un poco dolorosa, pero no se preocupe demasiado ya que realmente cobra sentido cuando recapacita sobre ello. Si trastea un poco con C#, descubrirá que realmente no parece que se necesite utilizar la palabra reservada `virtual` para un método. Si yo no la incluyo (y omito `override` también), el programa parece funcionar correctamente.

Esto se debe a que en esta situación no hay nada que sobreescribir, simplemente ha suministrado una nueva versión del método (de hecho, el compilador de C#, le mostrará un mensaje de advertencia donde se le avisa de que debe incluir la palabra reservada `new` para indicarlo):

```
public class CuentaInfantil : CuentaCliente, ICuenta
{
    public new bool RetirarEfectivo (decimal cantidad)
    {
       if (cantidad > 10)
       {
          return false;
       }
       if (saldo < cantidad)
       {
          return false;
       }
       saldo = saldo - cantidad;
       return true;
    }
}
```

Punto del Programador: No reemplace los métodos

Estoy muy en contra de reemplazar métodos en lugar de sobreescribirlos. Si desea tener una política que permita a los programadores crear versiones personalizadas de las clases de esta forma, es mucho más sensato hacer uso de la sobreescritura, Dado que esto permite una manera bien administrada de utilizar el método que ha anulado/invalidado. De hecho, me pregunto por qué estoy mencionando si quiera esta posibilidad...

4.9.5 Impedir y Detener la Sobreescritura

La sobreescritura es una característica del lenguaje muy poderosa. Esto significa que un programador puede cambiar una pequeña parte de la clase, y crear una nueva con todos los comportamientos del padre. En este proceso de diseño a medida que nos movemos hacia abajo a través del "árbol genealógico" de las clases, todo se vuelve cada vez más específico.

Sin embargo, la sobreescritura/reemplazamiento no siempre son acciones deseadas. Considere el método `ObtenerSaldo`. Este método nunca va a necesitar ser reemplazado. Y, sin embargo, un programador travieso y alocado podría escribir su propio método, y anular o reemplazar el del padre:

```
public new decimal ObtenerSaldo()
{
   return 1000000;
}
```

Este es el equivalente bancario de la botella de cerveza que nunca está vacía. No importa cuánto dinero se retire, ¡siempre devuelve un valor de saldo de un millón de libras!

Un programador travieso y alocado podría introducir el código anterior dentro de una clase y permitirse una oleada de gastos. Lo que significa que necesitamos una forma de marcar algunos métodos para que no puedan ser anulados o invalidados. C# nos permite hacer esto, utilizando el modificador `sealed` que significa “No puede anular o invalidar más este método”.

Desafortunadamente, este modificador es bastante complejo de utilizar. La regla es que solamente se puede sellar un método reemplazado (lo que significa que nosotros no podemos sellar el método virtual `ObtenerSaldo` en la clase `CuentaCliente`), y una persona traviesa y alocada siempre podría reemplazar un método sellado en el padre, con un método que tenga el mismo nombre en la hija.

El modificador `sealed`, en una declaración de clase, se utiliza para evitar que la clase se herede o pueda ser extendida, es decir, para que no puede utilizarse como base para otra clase.

```
public sealed class CuentaInfantil : CuentaCliente,ICuenta
{
    ...
}
```

El compilador ahora impedirá que `CuentaInfantil` se use como base de otra cuenta.

NOTAS BANCARIAS: PROTEJA SU CÓDIGO

En lo que respecta a la aplicación bancaria, el cliente no tendrá opiniones particularmente sólidas sobre cuándo debe utilizar elementos marcados como `sealed` en sus programas. No obstante, ellos querrán tener plena confianza en el código que usted escriba. Una de las cosas desafortunadas de este negocio, es que deberá admitir el hecho de que las personas que usan sus componentes pueden no ser del todo confiables. Esto significa que debe tomar medidas al diseñar el programa para decidir si los métodos deben ser declarados como virtuales, y también para asegurarse de sellar los elementos cuando tenga la necesidad de hacerlo.

Para un curso de programación a este nivel, es probable que yo esté haciendo demasiado énfasis y me esté extendiendo demasiado en trabajar este punto, y si no le encuentra sentido por ahora, no se preocupe, tan solamente recuerde que cuando esté desarrollando un programa este es otro riesgo que tendrá que tener en cuenta.

4.9.6 Constructores y Jerarquías

Un constructor es un método que toma el control durante el proceso de creación del objeto. Se utiliza para establecer los valores de inicialización de un objeto:

```
cuentaDeRob = new CuentaCliente("Rob", 100);
```

En este caso, queremos crear una nueva cuenta para Rob Miles con un saldo inicial de 100 libras.

El código anterior solamente funciona si la clase `CuentaCliente`, cuenta con un método constructor que acepte una cadena como primer parámetro y un valor decimal como segundo parámetro.

Podría pensar que puede resolverlo escribiendo un constructor como este:

```
public CuentaCliente(string enNombre, decimal enSaldo)
{
    nombre = enNombre;
    saldo = enSaldo;
}
```

Pero la clase `CuentaCliente` extiende de la clase `Cuenta`. En otras palabras, para crear una `CuentaCliente`, tengo que crear una `Cuenta`. Y la cuenta es la clase que contendrá un constructor que establece el nombre y el saldo inicial. En esta situación, el constructor en la clase hija, tendrá que llamar a un constructor en particular de la clase padre para establecerla antes de que sea creado. La palabra reservada `base` se utiliza para hacer una llamada al constructor padre. En otras palabras, la versión correcta del constructor de la cuenta cliente es la siguiente:

```
public CuentaCliente(string enNombre, decimal enSaldo) :
                                base (enNombre, enSaldo)
{
}
```

La palabra reservada `base` es utilizada del mismo modo en que `this` es utilizado para llamar a otro constructor de la misma clase. El constructor superior asume que la clase `Cuenta` que es hija de `CuentaCliente` cuenta con un constructor que acepta dos parámetros, el primero una cadena de texto y el segundo un valor decimal.

Encadenamiento de constructores

Al considerar los constructores y las jerarquías de las clases, debe recordar que para crear una instancia de una clase hija, primero debe crearse una instancia de la clase padre. Esto significa que el constructor en el padre, debe ejecutarse antes que el constructor en el hijo. En otras palabras, para crear una `CuentaCliente`, primero debe crear una `Cuenta`. Como consecuencia de este proceso, los programadores deben tener en cuenta el *encadenamiento de constructores*. Los programadores deben asegurarse de que, en cada nivel del proceso de creación, se llame a un constructor para establecer la clase a ese nivel.

Punto del Programador: Diseñe el proceso de construcción de sus clases

El medio por el cual se crean las instancias de su clase es algo que debe diseñar en el sistema que está construyendo. Esta forma parte de la arquitectura general del sistema que está construyendo. Piense en estos elementos como las vigas que se levantan para sostener los pisos y el techo de un gran edificio. Éstas indican a los programadores que van a construir los componentes que van a implementar la solución, cómo crear esos componentes. Por supuesto, es muy importante que tenga estos diseños anotados y disponibles para el equipo de desarrollo.

4.9.7 Métodos abstractos y clases

Por el momento, estamos utilizando la sobreescritura para modificar el comportamiento de un método padre existente. No obstante, también es posible utilizar la sobreescritura en un contexto ligeramente diferente. Podemos utilizarla para forzar una serie de comportamientos en los elementos de clase jerárquica. Si hay determinadas tareas o acciones que una cuenta deba hacer,

entonces podemos declararlas abstractas, y, a continuación, desde las clases hijas proporcionar realmente la implementación.
Por ejemplo, en el contexto de la aplicación bancaria, es posible que deseemos proporcionar un método que cree una carta de advertencia para aquellos clientes que tengan su cuenta al descubierto. Ésta tendrá que ser diferente para cada tipo de cuenta (nosotros no queremos utilizar el mismo lenguaje para una cuenta infantil que para una cuenta de un cliente de mayor edad). Esto significa que en el momento en que creamos el sistema de cuenta bancaria, nosotros sabemos que necesitamos este método, pero no sabemos exactamente lo que hace en cada situación.

Podríamos proporcionar un método "estándar" en la clase `CuentaCliente`, y confiar en que los programadores lo sobreescriban con un mensaje más específico, pero si lo hacemos así, no tenemos ninguna forma de asegurarnos de que ellos realmente vayan a proporcionar el método.

C# proporciona una forma de marcar un método como `abstract`. Esto significa que el cuerpo del método no es proporcionado en esta clase, sino que será proporcionado en una clase hija:

```
public abstract class Cuenta
{
    public abstract string CadenaCartaDesagradable();
}
```

El hecho de que mi nueva clase `Cuenta` contenga un método abstracto, significa que la clase en sí misma es `abstract` (y debe ser marcada como tal). No es posible crear una instancia de una clase abstracta. Si lo piensa con detenimiento, esto tiene su lógica. Una instancia de `Cuenta` no sabría qué hacer si el método `CadenaCartaDesagradable` fuese alguna vez llamado.
Una clase abstracta puede considerarse como un tipo de plantilla. Si quiere crear una instancia de una clase basada en un padre abstracto, debe proporcionar implementaciones de todos los métodos abstractos especificados en el padre.

Clases abstractas e interfaces

Puede determinar que una clase abstracta se parece mucho a una interfaz. Esto es cierto, ya que una interfaz también proporciona una "lista de compra" de métodos que deben ser proporcionados por una clase. Sin embargo, las clases abstractas son diferentes ya que pueden contener métodos completamente implementados junto con los abstractos. Esto puede ser útil ya que nos permite el no tener que implementar una y otra vez los mismos métodos en cada uno de los componentes que implementan una interfaz particular.

El problema es que sólo se puede heredar de un padre, por lo que solamente puede tomar los comportamientos de una clase. Si quiere también implementar interfaces, es posible que tenga que repetir los métodos también.

Tal vez en este punto, un ejemplo más completo podría ser de ayuda. Si consideramos nuestro problema de la cuenta bancaria, podemos identificar dos tipos de comportamiento:

- aquellos que cada tipo de cuenta bancaria debe proporcionar (por ejemplo, `IngresarEfectivo` y `ObtenerSaldo`)
- aquellos que cada tipo de cuenta bancaria debe proporcionar de una manera específica a ese tipo de cuenta en particular (por ejemplo, `RetirarEfectivo` y `CadenaCartaDesagradable`)

El truco es tomar todos los métodos incluidos en la primera categoría, e incluirlos dentro de la clase padre. Los métodos de la segunda categoría deben ser abstractos. Esto nos lleva a diseñar una clase como esta:

```
public interface ICuenta
{
   void IngresarEfectivo(decimal cantidad);
   bool RetirarEfectivo(decimal cantidad);
   decimal ObtenerSaldo();
   string CadenaCartaDesagradable();
}

public abstract class Cuenta : ICuenta
{
    private decimal saldo = 0;

    public abstract string CadenaCartaDesagradable();

    public virtual bool RetirarEfectivo(decimal cantidad)
    {
       if (saldo < cantidad)
       {
          return false;
       }
       saldo = saldo - cantidad;
       return true;
    }

    public decimal ObtenerSaldo()
    {
       return saldo;
    }

    public void IngresarEfectivo(decimal cantidad)
    {
       saldo = saldo + cantidad;
    }
}
```

```
public class CuentaCliente : Cuenta
{
    public override string CadenaCartaDesagradable()
    {
       return "Tiene la cuenta al descubierto";
    }
}

public class CuentaInfantil : Cuenta
{
    public override bool RetirarEfectivo(decimal cantidad)
    {
       if (saldo > 10)
       {
          return false;
       }
       return base.RetirarEfectivo(cantidad);
    }

    public override string CadenaCartaDesagradable()
    {
       return "Dile a papá que tienes la cuenta al descubierto";
    }
}
```

Código de Ejemplo 41 Utilizar un Método Abstracto

Este código requiere un estudio minucioso. Observe como yo he movido todas las acciones que todas las cuentas deben hacer dentro de la clase `Cuenta` padre. A continuación, he añadido métodos personalizados en el lugar que les corresponde dentro de las clases hijas. Tenga en cuenta también, que he dejado la interfaz en su lugar. Esto se debe a que; a pesar de que ahora tengo disponible esta estructura abstracta, todavía quiero pensar en los objetos de cuenta en términos de su "responsabilidad como cuenta", en lugar de en algún tipo específico particular.

Si creamos nuevos tipos de clases de cuenta basadas en la `Cuenta` padre, cada una de ellas, debe proporcionar su propia versión del método `CadenaCartaDesagradable`.

Referencias a clases abstractas

Las referencias a clases abstractas funcionan igual que las referencias a interfaces. Una referencia a una clase `Cuenta`, puede hacer referencia a cualquier clase que se extienda desde ese padre. Este principio podría parecer útil, ya que podemos considerar algo como una "cuenta" en lugar de como una `CuentaInfantil`.

Sin embargo, es preferible que gestione las referencias a elementos abstractos (como cuentas) en términos de su interfaz.

4.9.8 Diseño con Objetos y Componentes

Para el propósito de esta parte del texto, ahora cuenta con un amplio conocimiento de todos los recursos que pueden ser utilizados para diseñar grandes sistemas de software. Si tiene una compresión sólida de lo que una interfaz y una clase abstracta tienen por objeto ofrecerle, estará en una posición privilegiada en su camino para convertirse en un programador profesional. En términos generales:

Interface: le permite identificar una serie/conjunto de comportamientos comunes (es decir, métodos) que un componente puede implementar. Cualquier componente que implemente la interfaz puede ser pensado en términos de una referencia de ese tipo de interfaz. Un ejemplo concreto de esto sería algo como un método `IImprimirCopiaImpresa`. Muchos elementos en mi sistema bancario necesitarán realizar esta acción, por lo que podríamos exponer los detalles de comportamiento dentro de la interfaz para que cada uno de ellos los implementen a su manera según lo requieran. Entonces nuestra impresora sólo puede considerar cada una de las instancias que implementan esta interfaz exclusivamente de esta manera. Una vez que un componente puede implementar una interfaz, puede ser considerado puramente en términos de un componente con esta capacidad. Los objetos pueden implementar más de una interfaz, permitiéndoles a ellos presentar diferentes caras a los sistemas que los utilizan.

Abstract: le permite crear una clase padre que contiene una información de plantilla para todas las clases que la extienden/heredan. Si quiere crear un conjunto de elementos relacionados, por ejemplo, todos los diferentes tipos de cuentas bancarias con las que deba tratar, incluyendo tarjeta de crédito, cuenta de depósito, cuenta corriente, etc., entonces la mejor forma de hacer esto es estableciendo una clase padre que contenga métodos abstractos y no abstractos. Las clases hijas pueden hacer uso de los métodos del padre y anular aquellos que necesiten ser proporcionados de manera diferente para una clase en particular.

Utilizar Referencias de Interfaz

Una consideración importante es que incluso si hace uso de una clase padre abstracta, considero que todavía debe hacer uso de las interfaces para hacer referencia a los propios objetos de datos en sí. Esto le ofrece un grado de flexibilidad que puede utilizar a propósito para obtener un beneficio.

NOTAS BANCARIAS: HAGA UN BUEN USO DE INTERFACE Y ABSTRACT

Si nuestro banco se fusiona con otro banco y quiere compartir la información de sus cuentas, es posible que necesitemos una forma de utilizar sus cuentas. Si sus cuentas son también componentes de software (y deberían serlo), todo lo que tenemos que hacer es implementar las interfaces requeridas en cada extremo, y tras hacerlo, ambos sistemas podrán trabajar conjuntamente y comunicarse entre sí. En otras palabras, el otro banco debe crear los métodos en la interfaz `ICuenta`, obtener sus objetos de cuenta (como quiera que se llamen) para implementar la interfaz y, ¡listo !, yo habré cumplido el objetivo de utilizar sus cuentas.

Esto sería mucho más difícil y complejo, si todo mi sistema estuviese pensado en términos de una clase `Cuenta` padre, ya que sus clases no encajarían en esta jerarquía.

4.9.9 No entre en pánico

Todo lo que estamos viendo es demasiado profundo y enrevesado. Si no entiende todo ahora, no se preocupe. Estas características de C# están relacionadas con el proceso de diseño de software que es un modelo de negocio muy complejo. El punto importante a tener en cuenta, es que todas las características son proporcionadas para que pueda resolver un problema:

“Cree software empaquetado en componentes seguros e intercambiables.”

Las interfaces me permiten describir lo que cada componente puede hacer. Las jerarquías de clase me permiten reutilizar código dentro de esos componentes.

Y eso es todo.

4.10 Etiqueta Objeto

Ahora que ya hemos considerado los objetos "en general", y que sabemos cómo se pueden utilizar para diseñar e implementar grandes sistemas de software. Lo que tenemos que hacer ahora es echar un vistazo a otras características muy importantes, relacionadas con la forma en que usamos los objetos en nuestros programas.

4.10.1 Objetos y ToString

Podemos partir del hecho de que los objetos tienen una capacidad mágica para imprimirse. Si yo escribo el código:

```
int i = 99;
Console.WriteLine(i);
```

Se imprimirá por pantalla:

```
99;
```

El tipo de dato entero se imprime de alguna manera. Ahora es el momento de descubrir cómo lo logra, y también cómo podemos darles a nuestros propios objetos la misma capacidad mágica.

Resulta que todo esto es proporcionado por la "objetividad" de las cosas en C#. Nosotros sabemos que un objeto puede contener información y hacer cosas para nosotros (de hecho, nosotros hemos visto que la base del desarrollo de programas se encuentra en decidir lo que los objetos deben hacer y conseguir que hagan estas cosas). También hemos visto que podemos extender un objeto padre para crear un objeto hijo que tenga todas las capacidades del padre, más otras nuevas que nosotros queramos añadir. Ahora vamos a ver cómo estas capacidades que tienen los objetos son utilizadas para hacer que funcionen partes de la propia implementación de C#.

La clase Object

Cuando crea una nueva clase, esta no se crea realmente de la nada. De hecho, esta es hija de la clase `object`. En otras palabras, si escribo:

```
public class Cuenta {
```

- esto equivale a escribir:

```
public class Cuenta : object {
```

La clase `object` forma parte de C#, y todo deriva de la clase `object`. Esto tiene un par de consecuencias importantes:

- Cada uno de los objetos puede hacer, lo que un objeto puede hacer.
- Una referencia a un tipo objeto puede hacer referencia a cualquier clase.

Para el propósito que tenemos en estos momentos, el primer punto es el que tiene mayor importancia. Significa que todas las clases que se crean tienen una serie de comportamientos que heredan de su padre más remoto, el objeto. Si revisa el código fuente que hace que un objeto

funcione, encontrará un método llamado `ToString`. La implementación del método `ToString` en un objeto, devuelve una cadena con el nombre completo del tipo de ese objeto.

El método ToString

El sistema sabe que `ToString` está disponible para cada objeto, y si alguna vez necesita la versión de cadena de un objeto llamará a este método desde el objeto para obtener el texto.

En otras palabras:

```
object o = new object();
Console.WriteLine(o);
```

- imprimirá por pantalla:

```
System.Object
```

Puede llamar al método explícitamente si así lo desea:

```
Console.WriteLine(o.ToString());
```

- y el resultado que obtendrá será exactamente el mismo.

Una característica positiva del método `ToString` es que ha sido declarado `virtual`. Esto significa que podemos invalidarlo/anularlo para que se comporte como queramos:

```
class Cuenta
{
    private string nombre;
    private decimal saldo;

    public override string ToString()
    {
       return "Nombre: " + nombre + " saldo: " + saldo;
    }

    public Cuenta (string enNombre, decimal enSaldo)
    {
       nombre = enNombre;
       saldo = enSaldo;
    }
}
```

En la pequeña clase `Cuenta` vista arriba, he invalidado/anulado el método `ToString`, para que imprima el nombre del cliente y el valor del saldo. Esto significa que el código:

```
Cuenta a = new Cuenta("Rob", 25);
Console.WriteLine(a);
```

Código de Ejemplo 42 Metodo ToString

- imprimirá por pantalla:

```
Nombre: Rob saldo: 25
```

Así pues, desde el punto de vista protocolario, siempre que diseñe una clase, debe suministrar un método `ToString` para proporcionar una versión de texto del contenido de esa clase.

Obtener la descripción de cadena de un objeto padre

Si desea obtener una descripción de cadena del objeto padre, puede utilizar la palabra reservada de acceso `base` para conseguirlo. Esto a veces es útil si añade algún dato a una clase hija, y quiere imprimir por pantalla el contenido del padre primero:

```
public override string ToString()
{
   return base.ToString() + " Padre : " + nombreDelPadre;
}
```

El método anterior pertenece a una clase `CuentaHija`. Esta clase extiende de `CuentaCliente` y contiene el nombre del padre de la hija. El código anterior utiliza el método `ToString` en el objeto padre, y a continuación, coloca el nombre del padre al final antes de devolverlo. Lo bueno de esto es que, si el comportamiento de la clase padre cambia, es decir, se añaden nuevos miembros o el formato de la cadena cambia, la clase `CuentaHija` no tiene que cambiar, ya que solamente hará uso del método actualizado.

4.10.2 Objetos y comparaciones de igualdad

Nosotros hemos visto que comparar las referencias de objetos, no es lo mismo que hacerlo con sus valores. Sabemos que los objetos son gestionados a través de unas etiquetas con nombre, las cuales hacen referencia a elementos sin nombre guardados en alguna zona de la memoria. Cuando nosotros realizamos una comparación de igualdad, el sistema simplemente verifica si ambas referencias se encuentran en la misma localización de la memoria.

Para ver como esto podría causarnos un problema, podríamos considerar cómo implementaríamos los gráficos de un videojuego (dejamos apartado el banco por un momento). Los diversos objetos que forman parte de un videojuego, estarán localizados en una zona particular de la pantalla. Podemos expresar esta posición como una coordenada o punto, representadas por el valor x e y. Nosotros querremos comprobar si dos elementos (objetos) han colisionado.

```
class Punto
{
    public int x;
    public int y;
}
```

Esta es mi clase `Punto`. Yo ahora puedo crear instancias de `Punto`, y utilizarlas para gestionar mis objetos de juego:

```
Punto posicionNaveEspacial = new Punto();
posicionNaveEspacial.x = 1;
posicionNaveEspacial.y = 2;
Punto posicionMisil = new Punto();
posicionMisil.x = 1;
posicionMisil.y = 2;

if (posicionNaveEspacial == posicionMisil)
{
   Console.WriteLine("Bang");
}
```

Tenga en cuenta que yo he declarado los miembros `x` e `y` de la clase `Punto` con un modificador de acceso de tipo `public`. Esto es debido a que, en estos momentos, estoy más interesado en que mi programa se ejecute rápidamente que en protegerlos. El problema es que el código no funciona. A pesar de que he posicionado la nave espacial y el misil en el mismo lugar de la pantalla, la palabra Bang no se imprime.

Esto es debido a que, aunque los dos objetos `Punto` contienen los mismos datos, no están ubicados en la misma dirección de memoria, lo que significa que la comparación de igualdad fallará.

Punto del Programador: Asegúrese de usar la comparación correcta

Como programador, debe recordar que cuando se comparan objetos para verificar si tienen los mismos datos, debe utilizar el método `Equals` en lugar del operador `==`. Algunos de los errores/bugs más molestos que yo he tenido que corregir, han girado en torno a programadores que han confundido la comparación que tenían que realizar. Si nota que los elementos que contienen los mismos datos, no se están comparando de forma correcta, eche un vistazo a la comparación que está realizando.

Añadir su propio método Equals

La manera en la que puede resolver este problema, es proporcionando un método que pueda ser utilizado para comparar los dos puntos, y verificar si ellos están haciendo referencia a la misma

localización. Para hacer esto, debemos anular el comportamiento estándar de `Equals`, y añadir uno nuestro propio:

```
public override bool Equals(object obj)
{
   Punto p = (Punto obj);
   if (( p.x == x ) && ( p.y == y ))
   {
       return true;
   }
   else
   {
       return false;
   }
}
```

El método `Equals` recibe una referencia del elemento que se va a comparar. Tenga en cuenta que esta referencia es suministrada como referencia a un objeto. Lo primero que debemos hacer es crear una referencia a `Point`. Nosotros necesitamos hacer esto porque, queremos obtener los valores contenidos en x e y desde un `Point` (nosotros no podemos obtenerlos desde un objeto).

El objeto es convertido a `Point`, y a continuación, los valores x e y son comparados. Si ambos son iguales, el método devuelve `true`. Esto significa que yo puedo escribir un código como:

```
if (posicionMisil.Equals(posicionNaveEspacial))
{
    Console.WriteLine("Bang");
}
```

Código de Ejemplo 43 Método Equals

Y esta comparación funcionará, ya que ahora el método `Equals` compara el contenido de los dos puntos en lugar de las referencias.

Tenga en cuenta que realmente no necesito anular/invalidar el método `Equals`; Yo podría escribir uno llamado ElMismo que hiciera el trabajo. Sin embargo, el método `Equals` es en algunas ocasiones utilizado por métodos de la biblioteca C#, para verificar si dos objetos contienen los mismos datos, y, por lo tanto, la sobreescritura de `Equals` hace que mi clase se comporte correctamente en lo que a ellos respecta.

Punto del Programador: Los Comportamientos de Comparación de Igualdad son importantes

Para el director gerente de nuestra aplicación bancaria es muy importante que cada cuenta bancaria que se crea sea única. Si el banco alguna vez contiene dos cuentas idénticas, esto causaría serios problemas. El hecho de que todo en el sistema deba ser único podría llevarle a pensar que no hay necesidad de proporcionar una forma para comparar la igualdad de dos instancias de Cuenta. "Si el sistema nunca contendrá dos cuentas iguales, ¿qué necesidad existe de que yo pierda el tiempo escribiendo un código para compararlas?"

Sin embargo, existe una muy buena razón por la que podría encontrar de mucha utilidad el comparador `Equals`. Cuando escribe la parte del programa que almacena los datos y los vuelve a recuperar para tratarlos, encontrará que es muy útil tener una forma de comprobar que la información almacenada que está recuperando es idéntica a la que estableció. En esta situación, es muy importante disponer de un comportamiento de comparación de igualdad, y por eso pienso que debe proporcionar uno.

NOTAS BANCARIAS: LAS BUENAS PRACTICAS SON RECOMENDADAS

Se considera como una buena práctica, proporcionar los métodos `Equals` y `ToString` en las clases que construyamos. Esto permitirá que sus clases encajen con otras del sistema C#. También es muy útil hacer un uso adecuado de esto al escribir métodos en clases. Cuando comience a crear su sistema de administración de cuentas bancarias, deberá organizar una reunión de todos los programadores involucrados y hacerles saber que insistirá en seguir un desarrollo de buenas prácticas como este. De hecho, para un enfoque profesional, debe establecer normas que establezcan cuál de estos métodos va a proporcionar. Todos los programadores deben seguir estos estándares al participar en el proyecto.

4.10.3 Objetos y `this`

Por ahora ya debería de haberse hecho a la idea de que podemos utilizar una referencia a un objeto para acceder al contenido de los miembros de ese objeto. Considere:

```
public class Contador
{
    public int Dato=0;
    public void Contar()
    {
       Dato = Dato + 1;
    }
}
```

La clase anterior está formada por un único miembro de dato y un único método. El miembro de dato es un contador. Cada vez que se llama al método `Count`, el contador aumenta de valor. Puedo utilizar esta clase de la siguiente manera:

```
Contador c = new Contador();
c.Contar();
Console.WriteLine("Contador : " + c.Dato);
```

Esta llama al método y, a continuación, imprime el dato. Nosotros sabemos que en este contexto el operador `.` (punto) significa "sigue la referencia al objeto, y posteriormente utiliza este miembro de la clase ".

this como referencia a la instancia actual

Odio explicar esto. Yo quiero utilizar la palabra this para decir esto, pero esto también tiene un significado especial dentro de sus programas en C#. Quizás la mejor forma de salir airoso del problema es recordarle que cuando yo utilizo la palabra `this`, estoy indicando "una referencia a la instancia de ejecución actual de una clase". Cuando un método en una clase accede a una variable miembro, el compilador coloca automáticamente un `this.` delante de cada uso de un miembro de la clase. En otras palabras, la versión "apropiada" de la clase `Contador` sería la siguiente:

```
public class Contador
{
    public int Dato=0;
    public void Contar ()
    {
       this.Dato = this.Dato + 1;
    }
}
```

Nosotros podemos añadir un operador de referencia `this.` si queremos indicar de manera explícita que estamos utilizando un miembro de una clase en lugar de una variable local.

Pasar una referencia a sí misma a otra clase

Otra utilidad de `this` la encontramos cuando una instancia de clase necesita proporcionar una referencia a sí misma a otra clase que desea utilizarla. Una nueva cuenta bancaria podría almacenarse en un banco, pasando una referencia a sí misma a un método que la almacene:

```
banco.Almacenar(this);
```

Cuando el método `Almacenar` es llamado, se le proporciona una referencia a la instancia de cuenta actualmente en ejecución. El método `Almacenar` la acepta, y a continuación, la almacena de alguna forma. Por supuesto, en esta situación no estamos pasando una cuenta al Banco, sino que estamos pasando una referencia a la cuenta.

Si esto hace que le duela la cabeza, no se preocupe por ahora.

Confusion con this

Soy de los que considero que utilizar `this` es una buena practica, ya que las personas que leen mi código pueden saber al instante si estoy usando una variable local o un miembro de la clase. Sin embargo, también es cierto que puede llevar un poco de tiempo acostumbrarse a hacerlo. Si un miembro de mi clase contiene algunos métodos, y deseo usar uno de esos métodos, debo terminar escribiendo un código como este:

```
this.cuenta.EstablecerNombre("Rob");
```

Esto significa que "Yo tengo una variable miembro en esta clase llamada `cuenta`. Al realizar la llamada al método `EstablecerNombre` desde ese miembro establezco el nombre de la cuenta a "Rob".

4.11 El poder de las cadenas y los caracteres

Considero oportuno perder un poco más de tiempo conociendo en mayor detalle el tipo cadena. Este tipo de dato ofrece muchas características extra que merece la pena conocer, ya que le ahorrarán el tener que realizar muchas de las funcionalidades que necesitará utilizar en sus programas. Estas características son expuestas a través de métodos que puede llamar y utilizar sobre una referencia de cadena. Los métodos devolverán una nueva cadena, transformada de la forma en la que lo solicitó. Puede obtener toda la cadena convertida a mayúscula o minúscula,

eliminar todos los espacios en blanco existentes al inicio y/o al final de la cadena, y extraer sub-cadenas utilizando estos métodos.

4.11.1 Manipulación de cadenas

Las cadenas son un tanto especiales. Yo las considero un poco como murciélagos. Un murciélago puede ser considerado en según qué aspectos como un animal o un pájaro/ave. Una cadena puede ser considerada como un objeto (referenciadas por medio de una referencia) o como un valor (referenciadas por su valor).

Este comportamiento hibrido es proporcionado para hacernos la vida más fácil a los programadores. Por contra, también significa que tenemos que considerar a las cadenas como un elemento especial cuando estamos aprendiendo a programar, porque parecen romper algunas de las reglas que hemos aprendido hasta ahora.

Es muy importante que comprenda lo que sucede cuando transforma una cadena. Esto se debe a que, aunque las cadenas son objetos, en realidad no se comportan como tales todo el tiempo.

```
string s1 = "Rob";
string s2 = s1;
s2 = "diferente";
Console.WriteLine(s1 + " " + s2);
```

Si usted piensa que sabe de objetos y referencias, probablemente espere que `s1` cambie cuando `s2` sea cambiado. La segunda sentencia hace que la referencia `s2` apunte al mismo objeto que `s1`, por lo que al cambiar `s2` debería cambiar el objeto al que `s1` apunta también. Para obtener más información al respecto, consulte la sección 4.4.1.

Esto no ocurre como piensa, porque C# considera y trata a la instancia de un tipo cadena de una manera especial. Las cadenas son *inmutables*. Este concepto de inmutabilidad, se debe a que los programadores quieren que las cadenas se comporten como valores a este respecto.

4.11.2 Cadenas inmutables

La idea es que, si intenta cambiar una cadena, el sistema C# creará una nueva cadena y hará que la referencia que está "cambiando" apunte a la cadena modificada. En otras palabras, cuando el sistema llega a la línea:

```
s2 = "diferente";
```

- crea una nueva cadena que contiene el texto "diferente" y hace que `s2` apunte a ésta. La cadena a la que `s1` apunta no cambia. Así que, después de realizar la asignación `s1` y `s2` ya no apuntan al

mismo objeto. Este comportamiento, que nunca permite que un elemento sea modificado y que crea uno nuevo cada vez que se requiere, es la manera en que el sistema implementa el concepto de la inmutabilidad.

Usted podría preguntar: "¿Por qué se toma todo ese trabajo?" Podría parecer que todo este inconveniente podría evitarse simplemente creando cadenas de tipos de valor. Bueno, no. Considere el caso en el que esté almacenando un documento extenso en la memoria de la computadora. Al utilizar referencias, realmente solo necesitamos una instancia de cadena para la palabra “el” dentro de éste. Todas las ocurrencias de esa determinada palabra dentro del texto, solamente pueden hacer referencia a esa única instancia. Esto ahorra memoria y también hace que la búsqueda de palabras sea mucho más rápida.

4.11.3 Comparar cadenas

La propia naturaleza especial de las cadenas también le permite poder comparar cadenas utilizando el operador de comparación y conseguir el comportamiento que desea:

```
if ( s1 == s2 )
{
   Console.WriteLine("Iguales");
}
```

Si `s1` y `s2` fueran tipos de referencia adecuados, esta comparación solo se cumpliría si ambas hicieran referencia al mismo objeto. Pero en C# la comparación solo se cumple si ambas contienen el mismo texto. Puede utilizar el método `equals` si así lo prefiere:

```
if ( s1.Equals(s2) )
{
   Console.WriteLine("Todavía continúan siendo iguales");
}
```

4.11.4 Modificar cadenas

Puede leer caracteres individuales de una cadena indexándolos como lo haría con una matriz:

```
char primerCaracter = nombre[0];
```

Esta instrucción establecería en la variable `primerCaracter`, el primer carácter de la cadena. Sin embargo, no puede cambiar los caracteres del siguiente modo:

```
nombre[0] = "R";
```

- Al hacer esto, obtendría un error de compilación porque las cadenas son inmutables.

Puede extraer una secuencia de caracteres de una cadena utilizando el método `SubString`:

```
string s1 = "Rob";
s1=s1.Substring(1,2);
```

El primer parámetro indica la posición inicial, y el segundo establece el número de caracteres a copiar. La cadena resultante en `s1` tras realizar la operación sería `"ob"` (recuerde que las cadenas son indexadas desde la posición 0).

Puede omitir el segundo parámetro si lo desea, en cuyo caso se copian todos los caracteres hasta el final de la cadena:

```
string s1 = " Miles";
s1=s1.Substring(1,2);
```

- dejaría la cadena `"les"` en `s1`.

4.11.5 Longitud de la cadena

Todas las operaciones anteriores fallarán si prueba a hacer alguna operación que sobrepase el número de caracteres que contiene la cadena. En este sentido, las cadenas son como matrices. También es posible utilizar los mismos métodos sobre ellas para conocer la longitud de la cadena:

```
Console.WriteLine("Longitud: " + s1.Length);
```

La propiedad `Length` devuelve el número de caracteres que contiene la cadena.

4.11.6 Convertir cadena a mayúsculas y minúsculas

Se puede pedir a los objetos de tipo cadena que creen versiones nuevas y modificadas de sí mismas en formatos diferentes. Para ello, llame a los métodos sobre la referencia para conseguir el resultado que quiera:

```
s1=s1.ToUpper();
```

El método `ToUpper` devuelve una copia de la cadena con todas las letras convertidas a MAYÚSCULAS. También existe otro método inverso denominado `ToLower`.

4.11.7 Eliminar espacios en blanco al inicio o al final de la cadena

Otro método útil es `Trim`. Este método elimina cualquier espacio en blanco existente al inicio o al final de la cadena.

```
s1=s1.Trim();
```

Este método es útil si le preocupa que los usuarios de su aplicación puedan introducir un nombre que contenga espacios en blanco al inicio o al final de la cadena. Por ejemplo, un usuario podría introducir `"  Rob  "` en lugar de `"Rob"`. Si no elimina los espacios en blanco existentes, comprobará que las pruebas de comparación del nombre fallarán. Si lo desea, también tiene disponible por separado los métodos `TrimStart` y `TrimEnd` que permiten eliminar respectivamente, los espacios en blanco existentes al principio y al final de la cadena. Si utiliza el método `Trim` sobre una cadena que solamente contiene espacios en blanco, obtendrá una cadena sin caracteres (es decir, su longitud es cero).

4.11.8 Comandos de caracteres

La clase `char` también expone varios métodos muy útiles que puede utilizar para comprobar los valores de los caracteres individuales. Estos son métodos `static` que son invocados a través de la clase `char`, y que se utilizan para realizar comprobaciones de caracteres individuales en una variedad de formas:

char.IsDigit(carácter)	devuelve `true` si el carácter es un dígito (0 a 9)
char.IsLetter(carácter)	devuelve `true` si el carácter es una letra (a - z o A - Z)
char.IsLetterOrDigit(carácter)	devuelve `true` si el carácter es una letra o dígito
char.IsLower(carácter)	devuelve `true` si el carácter es una letra minúscula
char.IsUpper(carácter)	devuelve `true` si el carácter es una letra mayúscula
char.IsPunctuation(carácter)	devuelve `true` si el carácter es un signo de puntuación
char.IsWhiteSpace(carácter)	devuelve `true` si el carácter es un espacio, una tabulación o una nueva línea

Puede utilizar estos métodos cuando tenga que buscar en una cadena un carácter en particular.

4.11.9 Manipular cadenas mutables con la clase `StringBuilder`

Comprobará que es un verdadero sufrimiento el hecho de que no pueda asignar caracteres individuales a un tipo `string`. La razón de esta pesada molestia es que el tipo de dato `string` está realmente diseñado para contener cadenas, y no se proporciona una forma de editarlas. La alternativa consiste en utilizar StringBuilder, que es una clase de cadena mutable. Mutabilidad significa que, una vez creada una instancia de la clase, puede ser modificada mediante adición, eliminación, reemplazo o inserción de caracteres. La clase StringBuilder se encuentra en el espacio de nombres `System.Text`.

4.12 Propiedades

Las propiedades son útiles. Son un poco parecidas a la instrucción de selección `switch`, que nos permite a nosotros realizar una selección entre una serie de elementos fácilmente. Nosotros no necesitamos propiedades, pero C# las proporciona porque hacen que los programas sean un poco más fáciles de escribir y más sencillos de leer.

4.12.1 Propiedades como miembros de una clase

Una propiedad es un miembro de una clase que almacena un valor. Hemos visto que podemos utilizar una variable miembro para hacer este tipo de tareas, pero necesitamos hacer público el valor del miembro. Por ejemplo, el banco puede solicitarnos que llevemos un registro de los empleados. Uno de los elementos que quieren almacenar es la edad de los empleados. Puedo hacerlo de la siguiente forma:

```
public class Empleado
{
    public int Edad;
}
```

La clase contiene un miembro público. Puedo acceder al contenido de este miembro de la manera habitual:

```
Empleado s = new Empleado();
s.Edad = 21;
```

Puedo acceder a un miembro público de una clase directamente; simplemente indicando el nombre del miembro. El problema es que ya hemos comprobado que esta es una manera incorrecta de gestionar nuestros objetos. No hay nada que nos impida hacer algo como esto:

```
s.Edad = -100210232;
```

Esto puede ser peligroso, ya que como hemos visto en la línea anterior, al declarar el miembro `Edad` con el modificador de acceso `public`, no podemos detener este tipo de cambios.

4.12.2 Crear métodos Get y Set

Para tomar el control y permitir solamente acciones coherentes, podemos crear métodos de acceso `public`, que nos permitan obtener y establecer los datos de los miembros de nuestra clase, de una manera responsable y controlada. A continuación, podríamos establecer el miembro `Edad` mediante el modificador de acceso `private`, para que nadie pueda manipularlo:

```
public class Empleado
{
    public int edad;

    public int ObtenerEdad()
    {
        return this.edad;
    }

    public void EstablecerEdad( int enEdad )
    {
        if ( (enEdad > 0) && (enEdad < 120) )
        {
           this.edad = enEdad;
        }
    }
}
```

Ahora tenemos un control completo sobre nuestra miembro de dato, pero hemos tenido que escribir mucho código extra. Los programadores que ahora quieran realizar alguna acción con el valor de la edad, tienen que realizar llamadas a los métodos:

```
Empleado s = new Empleado();
s.EstablecerEdad(21);
Console.WriteLine( "La edad es : " + s.ObtenerEdad() );
```

Código de Ejemplo 44 Utilizar Propiedades

Podemos hacer que este código sea más sencillo, utilizando las propiedades y los descriptores de acceso `get` y `set`, que vamos a ver a continuación.

4.12.3 Utilizar Propiedades

Las propiedades son una forma de simplificar la gestión de datos como este. Una propiedad `edad` para la clase `Empleado` podría ser creada de la siguiente manera:

```
public class Empleado
{
    public int valorEdad;

    public int Edad
    {
        set
        {
           if ( (value > 0) && (value < 120) )
           {
              this.valorEdad = value;
           }
        }
        get
        {
           return this.valorEdad;
        }
    }
}
```

El valor de la edad ha sido ahora creado como una propiedad. Observe con detenimiento como hemos implementado en los descriptores de acceso `get` y `set`, la parte del código que obtiene y establece los datos de la propiedad. Lo bueno de las propiedades es que se utilizan justo como el miembro de la clase era:

```
Empleado s = new Empleado();
s.Edad = 21;
Console.WriteLine( "La edad es : " + s.Edad );
```

Cuando se tiene que asignar un valor a la propiedad `Edad`, se ejecuta el *descriptor de acceso* `set`. La palabra reservada `value` significa “lo que se está asignando”. Cuando la propiedad `Edad` tiene que ser leída, se ejecuta el código del *descriptor de acceso* `get`. Los descriptores de acceso nos proporcionan todas las ventajas de los métodos, siendo mucho más fáciles de utilizar.

Yo puedo hacer también otras cosas ingeniosas:

```
public int EdadEnMeses
{
    get
    {
       return this.valorEdad*12;
    }
}
```

Esta es una nueva propiedad, denominada `EdadEnMeses`. Esta propiedad solamente puede ser leída, dado que no presenta un comportamiento que permita otorgarle un valor. Sin embargo, ésta devuelve la edad en meses, basándose en el mismo valor de origen utilizado por la otra propiedad. Esto significa que puede proporcionar varias maneras diferentes de recuperar el mismo valor.

También puede proporcionar propiedades de sólo lectura omitiendo el comportamiento de asignación, es decir el descriptor de acceso `set`. Del mismo modo, puede proporcionar propiedades de sólo escritura omitiendo el comportamiento de lectura, es decir el descriptor de acceso `get`.

4.12.4 Propiedades e interfaces

Las interfaces describen un grupo de comportamientos relacionados. Las interfaces listan el conjunto de métodos que una clase siempre debe contener si la implementa. También es posible añadir propiedades a las interfaces. Esto se consigue, no incluyendo las instrucciones pertenecientes al cuerpo de los comportamientos de lectura y asignación:

```
Interface IEmpleado
{
    int Edad
    {
       get;
       set;
    }
}
```

Esta es una interfaz que especifica que la clase que la implementa, debe contener una propiedad `Edad` que tenga ambos comportamientos.

4.12.5 Problemas de Propiedad

Las propiedades son estupendas y nos permiten tener el código ordenado. Además, se utilizan en todas las clases de las bibliotecas .NET. Pero hay algunas cosas que debe tener en cuenta antes de decidir si usarlas o no:

Errores en la asignación de la propiedad

Ya hemos visto que el comportamiento de asignación `set`, puede rechazar valores si considera que están fuera de un rango. Esto está bien, pero no tiene una forma de comunicarle al usuario de la propiedad que esto ha sucedido. Según nuestra propiedad `Edad`, del código que establecimos anteriormente:

```
s.Edad = 121;
```

- fallaría. Sin embargo, la persona que realiza la asignación no sería consciente ni alertada de ello, ya que no hay forma de que la propiedad pueda devolver un valor que indique si la asignación se realizó correctamente o no. La única forma en la que el descriptor de acceso `set` de una propiedad, puede hacerlo sería lanzando una excepción, lo que no es considerado una buena forma de acabar.

Por otro lado, un método `EstablecerEdad`, podría devolver un valor que indique si funcionó correctamente o no:

```
public bool EstablecerEdad(int enEdad)
{
   if ((enEdad > 0) && (enEdad < 120))
   {
       this.edad = enEdad;
       return true;
   }
   return false;
}
```

Un programador que establezca el valor de la edad, puede ahora conocer si la asignación falló o no.

Si hay una situación en la que una asignación de propiedad puede fallar, yo **nunca** la expongo como una propiedad. Supongo que sería posible combinar un método de asignación y una propiedad de lectura, pero creo que sería una locura.

Las Propiedades ejecutan código

Cuando asigna un valor a una propiedad, en realidad está llamando a un método. Desafortunadamente, no hay ninguna manera en la que pueda indicarlo desde el código que realiza la asignación:

```
s.Edad = 99;
```

Esta línea de código anterior, parece muy inocente, pero podría ocurrir que mil líneas de código se ejecuten en el interior de la propiedad de asignación. Es posible utilizar esta característica como

una ventaja, ya que un objeto puede reaccionar y realizar un seguimiento de los cambios de propiedad. Pero esto también puede hacerlo muy confuso.

4.12.6 Utilizar Delegados

Los eventos y los delegados son una parte muy importante de C#. Por lo tanto, debe leer este texto cuidadosamente y asegurarse de que comprende todos los puntos que vamos a tratar.

Los delegados son útiles porque nos permiten manipular referencias a métodos. Podemos gestionar qué método particular va a ser llamado en una determinada situación en términos de un conjunto de datos que podemos mover.

Los delegados son determinantes para indicar cómo se gestionan los eventos en un programa de C#. Los eventos son determinados sucesos que ocurren, y a los que nuestro programa tiene que responder. Estos incluyen sucesos como hacer clic sobre los botones de la interfaz de usuario, la selección de un elemento en una lista desplegable, un temporizador que espera un intervalo de tiempo para desencadenar un tick, y mensajes que llegan a través de la red. En cada caso, necesitamos indicarle al sistema lo que debe hacer cuando se produzca el evento. La forma en que C# hace esto es permitiéndonos crear instancias de delegado, en las clases en las que nosotros ponemos los generadores de eventos. Cuando se genera el evento (en otros textos encontrará que se indica como: se "dispara" el evento), el método al que el evento hace referencia es invocado para entregar el mensaje. Yo defino un delegado como "Una forma de decirle a un fragmento de código del programa, lo que debe hacer cuando algo ocurra". Una descripción pija, podría tener la siguiente forma:

Un delegado es un tipo de referencia segura a un método de una clase.

Tipos de delegados seguros

El término *tipo de refencia segura* en este contexto significa que, si el método acepta dos parámetros enteros y devuelve una cadena de texto, el delegado para ese método tendrá exactamente la misma apariencia y no puede ser utilizado de ninguna otra manera. Esta palabra se utiliza para distinguir a un delegado de otros elementos como los punteros utilizados en lenguajes más primitivos como C.

En C, se pueden crear punteros a métodos, pero el entorno no sabe (o no se preocupa) del aspecto real de los métodos. Lo que significa que puede llamarlos de manera incorrecta y hacer que su programa explote. No se preocupe demasiado por esto. Tan solo recuerde que los delegados son seguros de utilizar.

Utilizar un Delegado

Como ejemplo, considere el cálculo de las tasas de nuestro banco. El banco tendrá una serie de métodos diferentes que realizan esta tarea, dependiendo del tipo de cuenta y del estado de cuenta del cliente. Es posible que desee tener una manera en que un programa pueda elegir qué método utilizar para el cálculo de las tasas a medida que se ejecuta.

Nosotros hemos visto que acciones como la sobreescritura, nos permiten crear métodos específicos para un particular tipo de objeto. Por ejemplo, nosotros proporcionamos un método `RetirarEfectivo` personalizado para la clase `CuentaInfantil`, que sólo nos permite retirar una cantidad limitada de dinero en efectivo. Estas técnicas son muy útiles, pero están duramente integradas dentro del código que escribimos. Una vez que haya compilado la clase, los métodos contenidos en ésta no pueden cambiar.

Un delegado es un sustituto de método. Al llamar al delegado, éste llama al método al que actualmente hace referencia. Esto significa que puedo usarlo como si fuese un tipo de selector de métodos. Un tipo delegado se crea así:

```
public delegate decimal CalcularTasa(decimal Saldo)
```

Tenga en cuenta que todavía no he creado ningún delegado, tan solo he comunicado al compilador el aspecto (denominado técnicamente "*firma*") que tiene el tipo delegado `CalcularTasa`. Este delegado puede sustituir a un método que acepte un solo parámetro decimal (el saldo de la cuenta), y que devuelva un valor decimal (la cantidad que vamos a cobrar al cliente). Un ejemplo de un método a considerar, que podríamos querer utilizar con este delegado es el siguiente:

```
public decimal TasaEstafa(decimal saldo)
{
   if (saldo < 0)
   {
       return 100;
   }
   else
   {
       return 1;
   }
}
```

Esta es una calculadora de tasas bastante malvada. Si la cuenta del cliente está al descubierto, la tasa es de 100 libras. Si tiene saldo en su cuenta, la tasa es de 1 libra. Si quiero utilizarla en mi programa. puedo crear una instancia de `CalcularTasa` que haga referencia a ésta:

```
CalcularTasa calc = new CalcularTasa(TasaEstafa);
```

Ahora, puedo "llamar" al delegado, y este realmente ejecutará el método `TasaEstafa` de la calculadora:

```
tasas = calc(100);
```

El delegado `calc` hace referencia a una instancia de delegado, que utilizará el método `TasaEstafa`. Puedo cambiar este comportamiento, creando otro delegado:

```
calc = new CalcularTasa(TasaAmistosa);
```

Ahora, cuando "llamo" al delegado `calc`, este ejecutará el método `TasaAmistosa`. Esto por supuesto, solo funciona si el método `TasaAmistosa` devuelve un valor de tipo decimal, y acepta un único valor decimal como parámetro.

```
using System;

public delegate decimal CalcularTasa(decimal saldo);

public class DemoDelegado
{
    public static decimal TasaEstafa(decimal saldo)
    {
        Console.WriteLine("Llamar al método Estafa");
        if (saldo < 0)
        {
           return 100;
        }
        else
        {
           return 1;
        }
    }

    public static decimal TasaAmistosa(decimal saldo)
    {
        Console.WriteLine("Llamar al método Amistoso");
        if (saldo < 0)
        {
           return 10;
        }
        else
        {
           return 1;
        }
    }
```

```
        public static void Main()
        {
            CalcularTasa calc;

            calc = new CalcularTasa(TasaEstafa);
            calc(-1); // esta instrucción llamará al método Estafa
            calc = new CalcularTasa(TasaAmistosa);
            calc(-1); // esta instrucción llamará al método Amistoso
            Console.ReadKey();
        }
    }
```

Código de Ejemplo 45 Utilizar Delegados

El ejemplo anterior muestra cómo todo encaja entre sí. El punto más importante y determinante a tener en cuenta, es que, en el método principal, hay dos llamadas del delegado `calc`. Cada una de estas llamadas realmente dará como resultado la ejecución de diferente código.
Una instancia de delegado es un objeto como cualquier otro. Esto significa que puede ser gestionada en términos de referencias, así que, puedo construir estructuras que contengan delegados, y también puedo pasar a los delegados dentro y fuera de los métodos. Puede considerar una lista de delegados como una lista de métodos a los que puedo llamar.

Esto nos da otra capa de abstracción y significa que ahora podemos diseñar programas que pueden tener comportamientos que cambian a medida que se ejecuta el programa. Sin embargo, por ahora, lo mejor es considerarlos en términos de cómo nosotros utilizamos los delegados para gestionar la respuesta de un programa a los eventos.

Punto del Programador: Utilice los Delegados con sensatez

Permitir a los programas cambiar lo que hacen mientras se están ejecutando, es algo bastante inusual de tener que hacer. Estoy presentando esto como un ejemplo de cómo los delegados permiten a un programa manipular referencias de métodos como objetos, más que como una buena forma de programar. Los delegados son muy utilizados para representar el método que controlará un evento que no tiene datos de evento, también para controlar un subproceso (como verá a continuación). Personalmente, no los uso mucho más allá que para esto, en los programas que desarrollo.

4.13 Crear un Sistema Bancario

En estos momentos, estamos en una situación en donde podemos crear cuentas bancarias que funcionen eficazmente. Podemos utilizar interfaces para definir el comportamiento de un componente de cuenta bancaria. Esto nos proporciona una forma de crear nuevos tipos de cuenta a medida que se desarrolla la actividad bancaria. Lo que ahora necesitamos es una forma de

almacenar un gran número de cuentas. Esta clase *contenedora* proporcionará métodos que nos permitan encontrar una cuenta particular basada en el nombre del titular. Podemos expresar el comportamiento que necesitamos de nuestro banco en términos de una interfaz:

```
Interface IBancaria
{
    ICuenta EncontrarCuenta(string nombre);
    bool AlmacenarCuenta(ICuenta cuenta);
}
```

Se puede utilizar una clase que implemente estos métodos para el almacenamiento de cuentas. Puedo establecer una cuenta dentro de ésta, y recuperarla posteriormente:

```
IBancaria bancoAmigo = new ArrayBancario(50);
ICuenta cuenta = new CuentaCliente("Rob", 0);
if (bancoAmigo. AlmacenarCuenta(cuenta)) {
    Console.WriteLine("Cuenta almacenada correctamente");
}
```

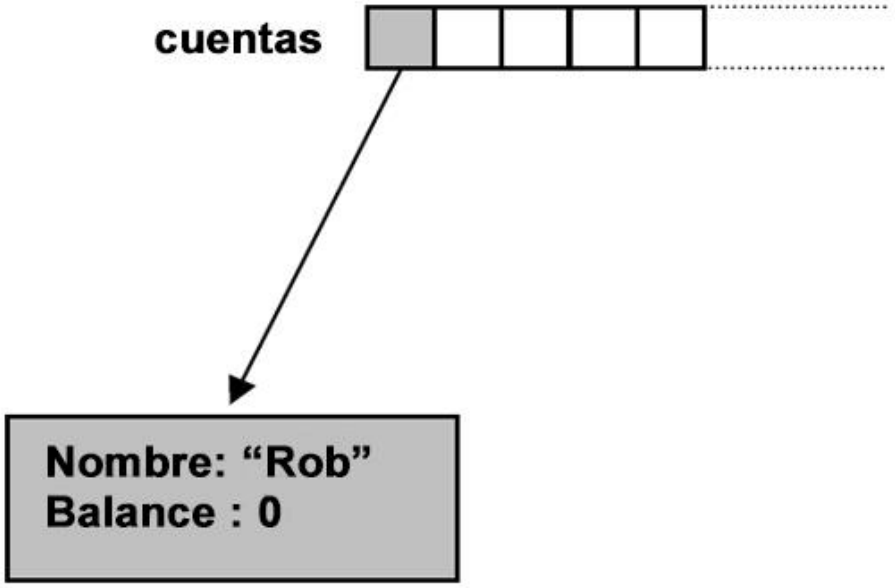
El código anterior crea un banco, y a continuación, establece una cuenta dentro de éste, e imprime un mensaje si la cuenta puedo ser guardada correctamente.

4.13.1 Almacenamiento de cuentas en una matriz

La clase `ArrayBancario` es un sistema de almacenamiento de cuentas bancarias que funciona utilizando matrices. Cuando usted crea una instancia de una matriz `ArrayBancario`, le indica a ésta el número de cuentas que desea almacenar a través del constructor. En el código anterior, estamos creando un banco con espacio suficiente para referencias a 50 cuentas. Al llamar al constructor, éste crea una matriz del tamaño apropiado:

```
private ICuenta[] cuentas;

public ArrayBancario(int tamanoBanco)
{
    cuentas = new ICuenta[tamanoBanco];
}
```

Es muy importante que comprenda lo que ha sucedido aquí. Nosotros no hemos creado ninguna cuenta. Lo que nosotros hemos creado es una matriz de referencias. Cada referencia en la matriz puede hacer referencia a un objeto que implemente la interfaz `ICuenta`. Por el momento, ninguna de las referencias apunta a alguna localización de memoria, todas están establecidas a `null`. Cuando agregamos una cuenta al banco, simplemente hacemos que una de las referencias apunte a esa instancia de cuenta. Nunca colocamos una cuenta "en" la matriz; en su lugar, establecemos una referencia a esa cuenta en la matriz.



El diagrama muestra el estado de la matriz de las cuentas, después de que hayamos almacenado nuestra primera cuenta. Los elementos de color claro están establecidos a `null`. El elemento al inicio de la matriz contiene una referencia a la instancia de la cuenta.

El método para añadir una cuenta a nuestro banco, tiene que encontrar la primera ubicación vacía en la matriz, y utilizarla para establecer la referencia a la cuenta que ha sido añadida:

```
public bool AlmacenarCuenta(ICuenta cuenta)
{
   int posicion = 0;
   for (posicion = 0; posicion < cuentas.Length; posicion++)
   {
       if (cuentas[posicion] == null)
       {
          cuentas[posicion] = cuenta;
          return true;
       }
   }
   return false;
}
```

Este método recorre la matriz buscando un elemento que contenga un valor `null`. Si lo encuentra, establece la referencia que apunta a la cuenta que se le ha pedido que almacene, y a continuación, devuelve `true`.
Si no encuentra ningún elemento `null` antes de terminar el recorrido de la matriz, devuelve `false` para indicar que el almacenamiento de la cuenta ha fallado.

Cuando queramos realizar alguna acción o tarea sobre una cuenta, tenemos antes que encontrarla. La interfaz del banco que hemos creado, provee un método denominado `EncontrarCuenta` que recorrerá la matriz de las referencias de cuenta para buscar la cuenta que coincida con un nombre de cliente en particular:

```
ICuenta cuentaEncontrada = arrayBancario.EncontrarCuenta("Rob");
```

El proceso devolverá la cuenta con el nombre requerido, o `null` si no se puede encontrar la cuenta. En la implementación basada en la matriz del banco, esto se consigue mediante una simple búsqueda:

```
public ICuenta EncontrarCuenta(string nombre)
{
   int posicion = 0;
   for (posicion = 0; posicion < cuentas.Length; posicion++)
   {
       if (cuentas[posicion] == null)
       {
          continue;
       }
       if (cuentas[posicion].ObtenerNombre() == nombre)
       {
          return cuentas[posicion];
       }
   }
   return false;
}
```

Este código recorre la matriz de cuentas bancarias buscando una entrada de un nombre que coincida con el que se está tratando de encontrar. Si encuentra un elemento que contiene una referencia `null`, se lo salta y pasa al siguiente. Si alcanza el final de la matriz sin haber encontrado una coincidencia, devolverá `null`, de lo contrario, devolverá una referencia a la cuenta que se encontró.

```
class ProgramaBancario
{
    public static void Main()
    {
        ArrayBanco nuestroBanco = new ArrayBancario(100);

        Cuenta nuevaCuenta = new Cuenta("Rob", "Casa de Rob", 1000000);

        if (nuestroBanco.AlmacenarCuenta(nuevaCuenta) == true)
             Console.WriteLine("Cuenta añadida al banco");

        ICuenta cuentaAlmacenada = nuestroBanco.EncontrarCuenta("Rob");
        if (cuentaAlmacenada != null)
             Console.WriteLine("Cuenta encontrada en el banco");
    }
}
```

Código de Ejemplo 46 Almacenar Cuentas en una Matriz

Este ejemplo crea una matriz denominada `ArrayBancario` que puede almacenar 100 cuentas. A continuación, crea una nueva cuenta y la guarda en el banco. A continuación, busca esa misma cuenta por su nombre y la encuentra.

4.13.2 Búsqueda y Rendimiento

La solución anterior funcionará correctamente, y podría utilizarse como base de una aplicación bancaria. Sin embargo, ésta se vuelve más lenta, a medida que aumenta el tamaño del banco. Cada vez que agregamos una nueva cuenta, la búsqueda se vuelve más lenta ya que el método `EncontrarCuenta`, debe realizar una búsqueda a través de más y más elementos para encontrarla. En un banco que tenga solo cincuenta cuentas esto no es un problema, pero en otro que tenga miles de cuentas, esta simple búsqueda será demasiado lenta.

4.13.3 Almacenar Cuentas utilizando una Tabla Hash

Afortunadamente para nosotros, podemos utilizar un recurso denominado "tabla hash", que nos permite encontrar fácilmente elementos basados en una clave. Esto es mucho más rápido que una búsqueda secuencial, ya que utiliza una técnica que nos llevará directamente al elemento requerido.

El planteamiento es que nosotros utilizamos una operación o función matemática (denominada "hashing") para la información de la propiedad de búsqueda, para generar un número que posteriormente puede ser utilizado para identificar la localización donde la información es almacenada.

Por ejemplo, podríamos coger todos los caracteres de la cadena del nombre de la cuenta, buscar el código ASCII de cada letra, y luego sumar estos valores. El nombre "Rob" podría convertirse a 82 + 111 + 98 = 291. Podríamos buscar en la localización 291 de nuestra cuenta.

Por supuesto, este código hash no es infalible, el nombre "Rpa" daría el mismo resultado (82 + 112 + 97) y se referiría a la misma localización. Cuando eso ocurre se dice poéticamente que se ha producido colisión de hasheo; o "*hash clash*" en la terminología anglosajona. Podemos hacer cosas ingeniosas con la forma en que combinamos los valores para reducir las probabilidades de que esto suceda, pero no podemos evitar por completo las colisiones.

Las colisiones de hasheo pueden resolverse añadiendo una comprobación cuando estamos almacenando la cuenta. Si descubrimos que la localización que nos gustaría utilizar no es nula, simplemente buscaremos la primera localización libre a partir de ese punto.

Cuando queremos encontrar un elemento dado, utilizamos el hash para que nos lleve directamente a la posición inicial de búsqueda, y a partir de esa localización, comenzamos la búsqueda del elemento que tenga el nombre correspondiente.

En resumen, la función hash nos da un punto de partida para nuestra búsqueda, y de esta forma nos evitamos el tener que recorrer toda la matriz de cuentas desde el inicio, como hacíamos en el código anterior. Esto acelera enormemente el acceso a los datos.

4.13.4 Utilizar la colección Hashtable

Afortunadamente para nosotros, los desarrolladores de C# han creado una tabla hash para que la utilicemos. Esto almacenará elementos para nosotros basados en un objeto particular que es denominado con el nombre de *clave*. Resulta muy fácil crear un mecanismo de almacenamiento bancario basado sobre esta característica:

```
class HashBancario : IBancaria
{
    Hashtable tablaHashBancaria = new Hashtable();

    public ICuenta EncontrarCuenta(string nombre)
    {
        return tablaHashBancaria[nombre] as ICuenta;
    }

    public bool AlmacenarCuenta(ICuenta cuenta)
    {
        tablaHashBancaria.Add(cuenta.ObtenerNombre(), cuenta);
        return true;
    }
}
```

Código de Ejemplo 47 Tabla Hash Bancaria

La clase `Hashtable` forma parte del espacio de nombres `System.Collections`. Ésta proporciona un método llamado `Add`, que suministra el valor de la clave y una referencia al elemento que se almacenará en ese código hash (en inglés denominado *hash code*). También le permite utilizar una referencia a la clave (en este caso el nombre) para localizar un elemento:

```
return tablaHashBancaria[nombre] as ICuenta;
```

Tenemos que utilizar la parte "`as`" en este código, porque la colección devolverá una referencia a un `object`. Nosotros queremos devolver una referencia a una instancia que implemente la interfaz `ICuenta`. El operador `as` se utiliza para realizar ciertos tipos de conversiones (forzamos al compilador a que considere a un elemento como un tipo de dato particular). Este es utilizado en preferencia al código:

```
return (ICuenta) tablaHashBancaria[nombre];
```

El operador `as` tiene la ventaja de que si `tablaHashBancaria`[nombre], no devuelve una cuenta, o devuelve un elemento del tipo incorrecto, el operador `as` generará una referencia `null`. Dado que esto es justamente lo que el llamador de nuestro método `EncontrarCuenta` está esperando, esto se puede devolver directamente. Si una conversion de tipo falla (es decir, en tiempo de ejecución la tabla hash bancaria devuelve el elemento incorrecto), nuestro programa fallará lanzando una excepción.

Resulta interesante indicar que debido a que hemos implementado nuestro comportamiento bancario usando una interfaz, podemos cambiar muy fácilmente la forma en que trabaja la parte de almacenamiento de la cuenta del programa sin cambiar nada más.

Notas bancarias: las propiedades clave son importantes

La acción que se está realizando aquí es exactamente la misma que sucede cuando usa su tarjeta de crédito para sacar dinero en efectivo del banco. El cajero automático utiliza información de la tarjeta como clave para encontrar el registro de su cuenta, de modo que pueda verificar el saldo de su cuenta y luego actualizarlo cuando se haya retirado el dinero.

En nuestro banco, nosotros solamente tenemos una simple clave, que es el nombre del titular de la cuenta. En un banco real la clave será más compleja, y puede haber más de una, de manera que puedan buscar un titular de cuenta por nombre, dirección o número de cuenta en función de la información que tengan disponibles.

Al recopilar metadatos sobre un sistema, a menudo deberá considerar cuáles de las propiedades de un elemento serán campos clave.

5 Programación Avanzada

5.1 Tipos Genéricos y Colecciones

Los tipos genéricos son muy útiles. Esta afirmación probablemente no le diga demasiado sobre éstos y lo que hacen, pero sí indica que son muy útiles. Estos tipos suenan un poco atemorizantes; dígale a la gente que aprendió cosas sobre "genéricos", y probablemente les evocará imágenes en su mente de personas con batas blancas y tubos de ensayo. Pero, estoy divagando demasiado, así que continuo por donde iba. Pienso que el concepto de Genéricos es probablemente "general", ya que la idea de ellos es que especifique una operación de propósito general y luego la aplique en diferentes contextos de una manera apropiada para cada uno de ellos. Si esto le suena un poco como si yo le estuviera hablando de los conceptos de abstracción y herencia, va por buen camino. Si no es así, entonces quizás valga la pena que vuelva a leer esas partes del libro destinadas a esos conceptos, hasta que le encuentre sentido a lo indicado en la introducción de este capítulo.

Quizás la mejor manera de hablar sobre los genéricos es ver cómo pueden ayudarnos a resolver un problema para nuestro sistema bancario. Acabamos de ver que podemos almacenar referencias de Cuentas en una matriz. Pero sabemos que cuando se crea una matriz, el programador debe especificar exactamente cuántos elementos contiene. Esto nos lleva a "la gota que colma el vaso", porque como el tamaño de la matriz del banco se estableció en 10.000, al agregar al cliente número 10.001, nuestro programa *crashea* (deja de funcionar de la forma en que se espera, o directamente deja de responder).

Una forma de resolver este problema es utilizando una matriz realmente grande, pero esta solución no nos interesa demasiado ya que para los bancos más pequeños el programa podría estar desperdiciando una gran cantidad de memoria. Afortunadamente, la biblioteca de clases de C# proporciona una serie de soluciones, comenzando por la clase `ArrayList`.

5.1.1 La clase ArrayList

La clase `ArrayList` es prima de `HashTable` que hemos visto anteriormente, ya que conviven en el mismo espacio de nombre `System.Collections`. Esta nos permite crear algo muy útil, una matriz que puede aumentar de tamaño dinámicamente. Siempre que lo necesitemos podremos añadir nuevos elementos a un `ArrayList`, y un código muy inteligente en la biblioteca se asegurará de que esto funciona correctamente.

Crear un ArrayList

Es muy fácil crear un `ArrayList`:

```
ArrayList almacen = new ArrayList();
```

Tenga en cuenta que no tiene que establecer el tamaño de un `ArrayList`, aunque hay constructores sobrecargados que le permiten especificar esta información y agilizar el código de la biblioteca un poco más:

```
ArrayList almacenarCincuenta = new ArrayList(50);
```

El `ArrayList` llamado `almacenarCincuenta`, inicialmente puede almacenar 50 referencias, pero puede contener una cantidad menor o superior según sea requerido.

Añadir Elementos a un ArrayList

Añadir elementos a un `ArrayList` es también muy fácil. La clase proporciona un método `Add`:

```
Cuenta cuentaDeRob = new Cuenta();
almacen.Add(cuentaDeRob);
```

Es importante recordar lo que está sucediendo aquí. No estamos estableciendo una `Cuenta` dentro del arraylist; estamos creando un elemento del arraylist apuntando (haciendo referencia) a esa cuenta. En este sentido, la palabra reservada `Add` puede ser un poco engañosa, ya que lo que realmente hace es añadir una referencia, no el elemento en sí.

Recuerde que sería perfectamente posible tener `cuentasDeRob` en múltiples arraylists, del mismo modo que su nombre puede aparecer en múltiples listados en el mundo real. Es posible que el banco tenga muchos tipos de listados de clientes. Tendrá una lista de todos los clientes, junto con una lista de clientes "especiales" y tal vez otra lista de quienes les deban más dinero. Por lo tanto, un elemento "en" un arraylist nunca está realmente dentro de ésta, el listado realmente contiene una referencia al elemento. Cuando un elemento se elimina de un arraylist, no es necesariamente destruido; simplemente ya no aparece en esa lista.

Acceder a los Elementos de un ArrayList

Se puede acceder a los elementos de un arraylists, de la misma forma que a los elementos de una matriz, pero con un giro algo más complejo. Sería de mucha ayuda, si pudiéramos obtener el contenido de la cuenta desde el arraylist y utilizarlo.

```
Cuenta a = almacen[0];
a.IngresarEfectivo(50);
```

Desafortunadamente, este código no funcionará. Si lo escribe, obtendrá un error de compilación. La razón de esto es que un ArrayList contiene una lista de referencias `object` (referencias de objeto). Si lo piensa, este es el único elemento que podría contener. Los desarrolladores de la clase `ArrayList`, no pueden saber con precisión el tipo de `object` que un programador va a querer utilizar, y esta es la razón, por la que ellos tuvieron que utilizar la referencia del objeto.

Sin duda, recordará el tema de las jerarquías de objetos donde una referencia de objeto puede apuntar a una instancia de cualquier clase (ya que todos ellos derivan de la clase base `object`) así que este es el único tipo de referencia con el que trabaja el arraylist.

Esto no es realmente un gran problema, porque un programa puede realizar una conversión de tipo, para convertir el tipo de elemento que devuelve un arraylist:

```
Cuenta a = (Cuenta) almacen[0];
a.IngresarEfectivo(50);
```

Este código obtendría el primer elemento del arraylist, lo convertiría en una clase `Cuenta`, y a continuación, ingresaría cincuenta libras dentro de ésta.

Un problema relativamente más grande es que un arraylist no es *typesafe*. Esto significa que no puedo tener la seguridad de que un arraylist que me sea proporcionado, nada más que tenga cuentas dentro de éste:

```
FregaderoDeCocina fc = new FregaderoDeCocina();
almacen.Add(fc);
```

Este código añade una referencia a una instancia `FregaderoDeCocina` en nuestro sistema de almacenamiento bancario. Esta acción podría causar problemas (y producirse una excepción), si en alguna ocasión trataramos de utilizarlo como una `Cuenta`. Para obtener un almacenamiento seguro de tipos de `Cuenta`, examinaremos con mayor detalle a los tipos genéricos un poco más adelante.

Eliminar Elementos de un ArrayList

No es realmente posible eliminar elementos de una matriz, pero la clase arraylist proporciona el comportamiento `Remove` que es realmente útil. Este comportamiento espera que se le proporcione la referencia del elemento a eliminar:

```
almacen.Remove(cuentaDeRob);
```

Esta línea de código elimina la **primera** ocurrencia de la referencia `cuentaDeRob` del arraylist `almacen`. Si `almacen` contenía más de una referencia a la `cuentaDeRob`, entonces cada uno de ellos podrían ser eliminadas individualmente. Tenga en cuenta que si la referencia proporcionada no está realmente en el arraylist (es decir, si ejecuto el código anterior, antes de almacenar la `cuentaDeRob`) esto no causa un error. Al eliminar un elemento, el tamaño del arraylist es decrementado.

Conocer el tamaño de un ArrayList

Puede utilizar la propiedad `Count` para conocer cuántos elementos hay en la lista:

```
if (almacen.Count == 0)
{
    Console.WriteLine("El banco está vacío");
}
```

Comprobar si un ArrayList contiene un elemento

El truco final que voy a mencionar (aunque hay muchas más cosas que un arraylist puede hacer) es el método `Contains`. Esta es simplemente una manera rápida de averiguar si un arraylist contiene o no una referencia particular.

```
if (a.Contains(cuentaDeRob))
{
    Console.WriteLine("Rob se encuentra dentro del sistema bancario");
}
```

El método `Contains` espera que se le pase una referencia a un objeto, y devuelve `true`, si el arraylist contiene esa referencia. Por supuesto, usted mismo también puede escribir este comportamiento, pero al tenerlo incorporado en la clase, el proceso se vuelve mucho más sencillo de realizar.

ArrayLists y Matrices

Los Arraylists y las matrices se parecen mucho. Ambas permiten almacenar una gran cantidad de elementos, y también permiten el uso de subíndices (los valores incluidos entre corchetes) para acceder a los elementos que contienen. Además, ambas lanzan excepciones si intenta acceder a elementos que no se encuentran en la matriz. Si lo desea, puede utilizar arraylists en lugar de matrices, de esta forma, tendrá la ventaja de disponer de un sistema de almacenamiento que aumentará o decrementará a medida que lo necesite, siempre teniendo en cuenta el problema de que un arraylist siempre contendrá referencias a objetos, y no a ninguna clase en particular. No obstante, esto se puede resolver utilizando la clase `List`.

5.1.2 La clase List

La clase `List` proporciona todo lo que ofrece un arraylist, con la ventaja añadida de que también es *typesafe*. La clase `List` es más actual que la clase `ArrayList`, ya que se basa en las características genéricas proporcionadas en una versión más reciente del lenguaje C#. Para entender cómo funciona, debemos introducirnos un poco en el tema de los genéricos.

Genéricos y Comportamientos

Si lo piensas, los comportamientos básicos de las matrices son siempre los mismos. Ya tenga una matriz de tipo `int`, `string`, `float` o de `Cuenta`, el trabajo que realiza es exactamente el mismo. Estas contienen un montón de elementos en un mismo lugar, y le proporciona algunos mecanismos para que trabaje con ellos. Las capacidades que ofrece una matriz de enteros son exactamente las mismas que ofrece para una matriz de `Cuentas`. Y dado que cada matriz es declarada para contener valores de un tipo particular, C# puede garantizar que una matriz siempre contiene elementos del tipo de valor adecuado. Debido a la forma en la que el sistema funciona, no existe una forma en la que pueda tomar una referencia de un elemento de tipo `Cuenta`, e introducirla en una matriz de elementos de tipo entero. Todo esto es positivo y beneficioso para nuestros intereses, pero la clase `ArrayList` quebranta algunas reglas.

La clase `ArrayList` fue incorporada a la biblioteca C# más tarde, y dado que no es una parte tan importante del lenguaje como lo es una matriz, esta se ve expuesta a ponerse en una situación comprometida, ya que permite almacenar referencias a cualquier tipo de objeto en un programa. Esto lo hace utilizando referencias, lo que puede dar a lugar a programas inseguros, ya que no hay nada que impida que puedan ser añadidas referencias de cualquier tipo a un arraylist.

Si usted me proporciona a mí una matriz de `Cuentas`, yo puedo estar absolutamente seguro de que todo en la matriz es una cuenta. Sin embargo, si usted me proporciona a mí un `ArrayList` no hay forma de que, yo pueda estar seguro de que todo lo que ésta contiene sean cuentas. Esta también podría contener un montón de referencias `FregaderoDeCocina`. Y yo solamente me empezaré a dar cuenta cuando empiece a procesar a los elementos como si fuesen `Cuentas` y mi programa comience a fallar.

Una forma de resolver este problema, habría sido encontrar una manera de hacer el `ArrayList` fuertemente tipado, para que se convierta en una parte tan importante de C# como la matriz. Sin embargo, esto no es lo que los diseñadores de C# consideraron. En su lugar, ellos introdujeron una nueva característica al lenguaje, los genéricos.
Los genéricos nos permiten escribir código que trate a los objetos como "elementos de un tipo particular". No importa con lo que sea que esté tratando, podemos solucionarlo cuando nosotros realmente queramos trabajar con alguna cosa.

Esto suena un poco confuso, ¿qué tal si vemos un ejemplo?

Si quisiera compartir canicas entre sus amigos, podría encontrar la forma de hacerlo. Algo así como: "Dividir el número de canicas entre el número de amigos, y posteriormente repartir las que sobren al azar". Ahora ya tiene un sistema que le permite compartir canicas. Por supuesto, este sistema también se puede utilizar para compartir dulces, pasteles o incluso coches. Cómo el sistema funciona no es particularmente importante. Usted puede tomar su algoritmo de intercambio universal, y usarlo para compartir casi cualquier cosa. El algoritmo de intercambio podría haber sido ideado sin la necesidad de preocuparse por el tipo de cosas que se comparten. A este respecto, los genéricos pueden considerarse como otra forma de abstracción.

Genéricos y la Lista

En el caso de los genéricos, el comportamiento que deseamos tener es el de una lista. Lo que la lista almacena no es importante al crear la lista, siempre que se tenga una forma de decirle lo que almacenar. Las características de C# que proporcionan genéricos añaden algunas nuevas notaciones para que pueda expresar esto. La clase `List` está definida en el espacio de nombre `System.Collections.Generic` y funciona de la siguiente manera:

```
List<Cuenta> listaDeCuentas = new List<Cuenta>();
```

La instrucción anterior crea una lista denominada `listaDeCuentas`, que puede almacenar referencias a `Cuentas`. El tipo presentado entre los caracteres `<` y `>` es como nosotros indicamos a la lista el tipo de cosas que puede almacenar. Nosotros podríamos crear una `List` que pueda almacenar enteros de la misma forma:

```
List<int> puntuaciones = new List<int>();
```

Dado que le hemos indicado al compilador el tipo de cosas que puede contener la lista, éste puede realizar la validación de tipos y asegurarse de que no va ocurrir ningún imprevisto cuando se utiliza la lista:

```
FregaderoDeCocina fc = new FregaderoDeCocina();
listaDeCuentas.Add(k);
```

Las instrucciones anteriores, causarían un error de compilación, dado que la variable `listaDeCuentas` es declarada como una lista de almacenamiento de referencias de `Cuenta`, y no aceptará el fregadero de la cocina.

Dado que el compilador sabe que `listaDeCuentas` almacena referencias `Cuenta`, esto significa que puede escribir un código como este:

```
listaDeCuentas[0].IngresarEfectivo(50);
```

No es necesario convertir el elemento de la lista ya que el tipo ya se ha establecido.

Usted puede hacer todo lo que realiza con una `List` (`Add`, `Count`, `Remove`), con un `ArrayList`, lo que lo convierte en la manera perfecta de almacenar un gran número de elementos de un tipo determinado.

5.1.3 La clase Dictionary

Así como `ArrayList` tiene un primo más poderoso, genéricamente mejorado, llamado `List`, también un `HashTable` tiene un primo más poderoso llamado `Dictionary`. Esto permite que la

clave de su tabla hash, y los elementos que esta almacena, sean de tipo seguro. Por ejemplo, es posible que queramos utilizar el nombre de un titular de cuenta como una forma de localizar una cuenta particular. En otras palabras, nosotros tenemos una cadena como la clave y una referencia de `Cuenta` como el valor. Podemos crear un diccionario para mantener estos pares clave/valor, de la siguiente manera:

```
Dictionary<string,Cuenta> diccionarioDeCuentas =
      new Dictionary<string,Cuenta>();
```

Nosotros ahora podemos añadir elementos a nuestro diccionario:

```
diccionarioDeCuentas.Add("Rob", cuentaDeRob);
```

Esto se parece a la forma en la que utilizamos el `HashTable` (y lo es). Sin embargo, el uso de la clase `Dictionary` tiene la ventaja de que nosotros solamente podemos añadir valores de `Cuenta` estableciéndolos a través de una cadena. En otras palabras, instrucciones como las siguientes serían rechazadas.

```
FregaderoDeCocina fc = new FregaderoDeCocina();
diccionarioDeCuentas.Add("Gluglú", fc);
```

Una ventaja adicional es que no necesitamos convertir los resultados:

```
d["Rob"].IngresarEfectivo(50);
```

Este código se encargaría de encontrar el elemento que tiene la clave hash "Rob", y a continuación, añadirá cincuenta libras al valor de la cuenta. El único problema que encontramos al utilizar este diccionario, es que, si no hay ningún elemento con la clave "Rob", al fallar el intento por encontrarlo será lanzada una `KeyNotFoundException`. Puede evitar esto preguntando al diccionario si contiene una clave en particular:

```
if (d.ContainsKey("Rob"))
{
    Console.WriteLine("Rob se encuentra en el sistema bancario");
}
```

El método devuelve `true` si la clave es encontrada. Tenga en cuenta que también necesitará usar este método para asegurarse de que haya una clave en el diccionario antes de agregar un nuevo elemento, porque un diccionario no permitirá que dos elementos tengan la misma clave.

```
class DiccionarioBancario
{
    Dictionary<string,ICuenta> diccionarioDeCuentas =
                    new Dictionary<string,ICuenta>();

    public ICuenta EncontrarCuenta(string nombre)
    {
        if (diccionarioDeCuentas.ContainsKey(nombre))
            return diccionarioDeCuentas[nombre];
        else
            return null;
    }

    public bool AlmacenarCuenta(ICuenta cuenta)
    {
        if (diccionarioDeCuentas.ContainsKey(cuenta.ObtenerNombre()))
            return false;

        diccionarioDeCuentas.Add(cuenta.ObtenerNombre(), cuenta);
        return true;
    }
}
```

Código de Ejemplo 48 Diccionario Bancario

El código anterior muestra cómo podemos crear los métodos `EncontrarCuenta` y `AlmacenarCuenta` utilizado un diccionario. Los métodos realizan comprobaciones para asegurarse de que el diccionario se utiliza correctamente.

La clase `Dictionary` es maravillosa para almacenar colecciones de pares que contienen claves y valores de una forma segura, y le recomiendo utilizarla encarecidamente.

5.1.4 Escribir Código Genérico

Las clases `List` y `Dictionary`, fueron por supuesto escritas en C# y hacen uso de las características genéricas del lenguaje. El cómo se utilizan estas características para crear clases genéricas va un poco más allá del alcance de este texto, pero le recomiendo encarecidamente que busque más información sobre los genéricos, ya que puede hacer que crear código, en particular de rutinas de biblioteca, sea mucho más fácil.

Punto del Programador: Utilice los genéricos Dictionary y List

Realmente uno no tiene que saberlo todo sobre los genéricos, para poder apreciar cuán útiles son estos dos elementos. Cuando necesite almacenar una gran cantidad de cosas, construya una `List`. Si necesita realizar búsquedas de elementos en base a una clave, utilice un Dictionary. No se preocupe por el rendimiento. Los programadores que desarrollaron estos elementos son muy inteligentes. Usted debería utilizar estos elementos en preferencia a los hashtables y arraylists.

5.2 Almacenar Objetos de Negocio

Para que nuestro banco realmente funcione, tenemos que contar con una forma de almacenar las cuentas bancarias y recuperarlas cuando lo necesitemos. Si queremos que nuestro programa pueda procesar una gran cantidad de cuentas, sabemos que tenemos que crear una matriz para conseguirlo. Esto significa que nuestros requerimientos de almacenamiento de datos son que debemos cargar todas las cuentas en la memoria cuando el programa se inicie y, a continuación, guardarlos cuando se cierre el programa.

Para empezar, consideraremos cómo guardar una cuenta. A continuación, pasaremos a considerar el código que nos permitirá guardar un gran número de ellas. La cuenta con la que vamos a trabajar solo tiene dos miembros, pero las ideas que vamos a examinar y probar pueden ser ampliadas para manejar clases que contengan cantidades de datos mucho mayores. Podemos expresar el comportamiento requerido de la cuenta en términos de la siguiente interfaz:

```
public interface ICuenta
{
   void IngresarEfectivo( decimal cantidad );
   bool RetirarEfectivo( decimal cantidad );
   decimal ObtenerSaldo();
   string ObtenerNombre();
}
```

Todas las cuentas del banco son gestionadas en términos de objetos, que implementan esta interfaz para gestionar el saldo y leer el nombre del propietario de la cuenta.

Nosotros podemos crear una clase `CuentaCliente` que implemente la interfaz anterior y contenga los métodos requeridos. Esta también contará con un método constructor que permita establecer los valores de inicialización para el nombre y el saldo de la cuenta que serán establecidos cuando se crea la cuenta.

```
public class CuentaCliente : ICuenta
{
    public CuentaCliente(
       string nuevoNombre,
       decimal saldoInicial)
    {
       private decimal saldo = 0;
       private string nombre;
    }

    public virtual bool RetirarEfectivo( decimal cantidad )
    {
       if ( saldo < cantidad )
       {
           return false;
       }
       saldo = saldo - cantidad;
       return true;
    }

    public void IngresarEfectivo( decimal cantidad )
    {
       saldo = saldo + cantidad;
    }

    public decimal ObtenerSaldo()
    {
       return saldo;
    }

    public string ObtenerNombre()
    {
       return nombre;
    }

}
```

Tenga en cuenta que esta versión de la clase no realiza ninguna comprobación de errores de los valores de entrada, por lo que no es lo que yo llamaría código de "producción", pero nos sirve aquí para para ilustrar cómo se pueden guardar y restaurar los datos de la clase.

5.2.1 Guardar una Cuenta

La mejor manera de lograr el comportamiento de guardado, es hacer que una cuenta sea responsable de salvarse a sí misma. De hecho, si piensa en ello, esta es la única forma en que podemos guardar una cuenta, ya que cualquier mecanismo de guardado necesitaría guardar los

datos en la cuenta que es privada y que, por lo tanto, solo es visible para los métodos dentro de la clase.

Podemos añadir un método `Guardar` a nuestra clase `CuentaCliente`:

```
public bool Guardar (string nombredearchivo)
{
   try
   {
       System.IO.TextWriter textoSalida =
           new System.IO.StreamWriter(nombredearchivo);
       textoDeSalida.WriteLine(nombre);
       textoDeSalida.WriteLine(saldo);
       textoDeSalida.Close();
   }
   catch
   {
       return false;
   }
   return true;
}
```

Este método recibe el nombre del archivo en el que se guardará la cuenta, y escribe en su interior el nombre del cliente y el saldo de la cuenta. Entonces, podría hacer cosas como esta:

```
if (Rob.Guardar ("archivoDeSalida.txt"))
{
    Console.WriteLine ("Guardado correctamente");
}
```

Esto solicitaría que la cuenta a la que hace referencia Rob se guarde en un archivo llamado `"archivoDeSalida.txt"`. Tenga en cuenta que he escrito el código de modo que si el resultado del archivo falla el método devuelva `false`, para indicar que la acción de guardado no fue realizada correctamente.

5.2.2 Cargar una Cuenta

El proceso de carga es un poco más complicado que el de guardado. Cuando nosotros grabamos una cuenta, tenemos una cuenta en la que queremos grabar. Cuando nosotros cargamos, no hay una instancia de cuenta para cargar. Una forma de evitar esto es escribir un método estático que cree una cuenta con un nombre de archivo proporcionado:

```
public static CuentaCliente Cargar(string nombredearchivo)
{
   CuentaCliente resultado = null;
   System.IO.TextReader textoEn = null;

   try
   {
       textoEn = new System.IO.StreamReader(nombredearchivo);
       string textoNombre = textoEn.ReadLine();
       string textoSaldo = textEn.ReadLine();
       decimal saldo = decimal.Parse(textoSaldo);
       resultado = new CuentaCliente(textoNombre,saldo);
   }
   catch
   {
       return null;
   }
   finally
   {
       if (textoEn != null) textEn.Close();
   }
       return resultado;
}
```

Este método abre un archivo, recupera el valor del saldo y luego crea una nueva `CuentaCliente` con el valor del saldo y el nombre.

```
class DemoGuardar
{
    public static void Main()

    {
        CuentaCliente prueba = new CuentaCliente("Rob", 1000000);
        prueba.Guardar("Prueba.txt");
        CuentaCliente cargada = CuentaCliente.Cargar("Prueba.txt");
        Console.WriteLine(cargada.ObtenerNombre());
    }
}
```

Código de Ejemplo 49 Guardar y Cargar Cuenta

Este código de ejemplo, muestra cómo un programa puede guardar una cuenta en un archivo y, a continuación, cargarla de nuevo.

El método `Cargar` del código visto más arriba, garantiza una serie de cosas, si algo inesperado ocurre:

- No lanza ninguna excepción que el llamador deba capturar.
- Devuelve null para indicar que la carga falló, si ocurre algo inesperado.
- Se asegura siempre de cerrar el archivo que abre.

Esto es lo que yo llamaría un código de calidad "profesional", y es algo que debería tener como objetivo cuando escriba código que va posteriormente a vender.

Nosotros podemos utilizarlo de la siguiente manera:

```
prueba = CuentaCliente.Cargar("prueba.txt");
```

Este tipo de método a veces se denomina método de "fábrica", ya que crea una instancia de una clase para nosotros. Si el método de fábrica falla (porque el archivo no puede ser encontrado o no contiene contenido válido), devolverá un resultado null, el cual podemos testear:

```
if (prueba == null)
{
    Console.WriteLine ("Carga fallida");
}
```

Punto del Programador: Hay un límite dentro de lo que puede hacer

Tenga en cuenta que, si un programador incompetente incorporara mi método `Cargar` a su código, y se olvidara de comprobar el resultado que este método devuelve, el programa podría devolver una referencia nula si no se encuentra al cliente. Esto significa que su programa fallará en esta situación, y probablemente me responsabilizara de esto (cosa que sería muy injusta).

La forma en la que usted puede evitar esto es asegurarse de documentar en letras mayúsculas el comportamiento de retorno nulo, para que los usuarios tengan constancia de cómo se comporta el método. También se pueden utilizar herramientas de análisis de código que son un poco como "compiladores con actitud" (una buena herramienta destinada a esto es FxCop). Estas herramientas de análisis de código se utilizan en situaciones donde se ignoran los resultados de los métodos y los señalan como posibles errores. Sin embargo, no hay nada que pueda hacer ante usuarios incompetentes que utilizan tu software, así que solo tienes que asegurarte de que, pase lo que pase, tu parte no sea quebrantada.

En realidad, a medida que envejezco, me inclino más a hacer que mis programas generen excepciones en situaciones como esta, ya que esto asegura que un fallo del sistema, sea reconocido mucho antes. Sin embargo, pueden existir argumentos contrarios a este, que podemos discutir mejor en un pub.

5.2.3 Múltiples cuentas

El código anterior nos permite almacenar y cargar cuentas individuales. Sin embargo, hasta este instante, solamente podemos guardar una cuenta en un archivo. Podríamos crear un nuevo archivo para cada cuenta, pero esto sería confuso e ineficiente, debido a la gran cantidad de apertura y cierre de archivos que se requieren (teniendo en cuenta que nuestro banco puede almacenar miles de cuentas).

Utilizar streams

Una solución más adecuada es proporcionar al método de guardado de archivos un flujo para guardarse a sí mismo, en lugar de un nombre de archivo. Un *stream* es un objeto que la biblioteca C# crea cuando nosotros realizamos e iniciamos una apertura de conexión a un archivo:

```
System.IO.TextWriter textoDeSalida =
    new System.IO.StreamWriter("Prueba.txt");
```

La referencia `textoDeSalida` hace referencia a un flujo que está conectado al archivo `Prueba.txt`. Podemos crear un método de guardado que acepte la referencia de flujo como un parámetro en lugar de un nombre de archivo:

```
public void Guardar(System.IO.TextWriter textoDeSalida)
{
   textoDeSalida.WriteLine(nombre);
   textoDeSalida.WriteLine(saldo);
}
```

Este método `Guardar` puede ser llamado desde nuestro método original de guardado de archivo:

```
public bool Guardar(string nombredearchivo)
{
   System.IO.TextWriter textoDeSalida = null;
   try
   {
       textoDeSalida = new System.IO.StreamWriter(nombredearchivo);
       Guardar(textoDeSalida);
   }
   catch
   {
       return false;
   }
   finally
   {
       if (textoEn != null)
       {
           textEn.Close();
```

```
        }
    }
        return true;
}
```

Este método crea un stream y, a continuación, lo pasa al método de guardado para guardar el elemento. Tenga en cuenta que este es un ejemplo de *sobrecarga* en el que tenemos dos métodos que comparten el mismo nombre.

El método de carga para nuestra cuenta bancaria puede ser sustituido por uno que funcione de manera similar.

```
public static CuentaCliente Cargar(System.IO.TextReader textoEn)
{
    CuentaCliente resultado = null;

    try
    {
        string nombre = textoEn.ReadLine();
        string textoSaldo = textEn.ReadLine();
        decimal saldo = decimal.Parse(textoSaldo);
        resultado = new CuentaCliente(nombre,saldo);
    }
    catch
    {
        return null;
    }
    return resultado;
}
```

Este método es suministrado con una secuencia de texto, lee el nombre y el saldo desde la secuencia suministrada, y crea una nueva `CuentaCliente` basada en esta información.

Punto del Programador: Los Streams son maravillosos

Utilizar objetos streams es una muy buena idea. El sistema de entrada/salida de C# nos permite conectar secuencias a todo tipo de elementos, no solamente a archivos de disco. Por ejemplo, puede crear una secuencia que esté conectada a un puerto de red. Esto significa que si hace que sus objetos comerciales se guarden y se carguen utilizando streams, luego pueden ser enviados a través de conexiones de red sin ningún trabajo adicional por su parte.

Guardar y cargar cuentas bancarias

Ahora que tenemos una forma de guardar múltiples cuentas en una sola secuencia stream, podemos escribir un método de guardado para el banco. Este abrirá un flujo de datos y guardará toda la información de las cuentas:

```
public void Guardar(System.IO.TextWriter textoDeSalida)
{
   textoDeSalida.WriteLine(tablaHashBancaria.Count);
   foreach (CuentaCliente cuenta in tablaHashBancaria.Values)
   {
       cuenta.Guardar(textoDeSalida);
   }
```

Este es el método `Guardar` que podría ser añadido a nuestro `Hashtable` bancario. Éste recorre cada cuenta existente dentro de la table hash, y las guarda en la secuencia especificada. Debe tener en cuenta que en este caso estamos utilizando, una nueva construcción de bucle disponible en C#, llamada foreach. Esto es de mucha utilidad cuando nosotros tratamos con colecciones de datos. Este tipo de bucle trabaja a través de una *colección*, en este caso la propiedad `Values` de la `tablaHashBancaria`, y suministra cada elemento uno por uno.

También debe tener en cuenta que antes de escribir nada, este método escribe por pantalla el número de clientes en el banco. Este número se obtiene a través de la propiedad `Count` de la clase `Hashtable`. Nosotros hacemos esto para que cuando el banco vuelva a leer en el método de carga sepa cuántas cuentas se requieren. Podríamos simplemente escribir los datos y luego dejar que el método de carga se detenga cuando alcance el final del archivo, pero esto no nos permitiría detectar si el archivo ha sido acortado.

El método `Cargar` para el sistema bancario completo es el siguiente:

```
public static HashBancario Cargar(System.IO.TextReader textoEn)
{
   HashBancario resultado = new HashBancario();
   string contarCadena = textoEn.ReadLine();
   int contador = int.Parse(contarCadena);

   for (int i = 0; i < contador; i++)
   {
       CuenteCliente cuenta = CuentaCliente.Cargar(textoEn);
       resultado.tablaHashBancaria.Add(cuenta.ObtenerNombre(), cuenta);
   }
       return resultado;
}
```

Este lee el tamaño del banco, y a continuación, recorre todas las cuentas disponibles una por una, y las agrega a la tabla hash. Tenga en cuenta que este método de carga no gestiona los errores. Una versión del programa en fase de producción verificará que cada cuenta fue cargada correctamente antes de agregarla a la tabla hash.

```
class ProgramaBancario
{
    public static void Main()
    {
        DictionaryBancario nuestroBanco = new DictionaryBancario();

        Cuenta nuevaCuenta = new Cuenta("Rob", 1000000);

        if (nuestroBanco.AlmacenarCuenta(nuevaCuenta) == true)
             Console.WriteLine("Cuenta añadida al banco");

         nuestroBanco.Guardar("Prueba.txt");

        DictionaryBancario cargarBanco =
             DictionaryBancario.Cargar("Prueba.txt");

        ICuenta cuentaAlmacenada = nuestroBanco.EncontrarCuenta("Rob");

        if (cuentaAlmacenada != null)
                Console.WriteLine("Cuenta encontrada en el banco");
    }
}
```

Código de Ejemplo 50 Guardar un banco completo

Este código de ejemplo muestra como trabaja un banco utilizando una clase Dictionary. El ejemplo anterior solo almacena una cuenta, pero podría ampliarse para almacenar tantas como sea necesario.

Notas Bancarias: Almacenamiento de Datos a Gran Escala

Lo que nosotros hemos hecho es crear una forma en la que podemos almacenar un gran número de valores de cuentas bancarias en un solo archivo. Lo hemos hecho sin sacrificar ninguna cohesión, ya que sólo la clase `CuentaCliente` es responsable del contenido. También hemos puesto especial atención en asegurarnos de que cada vez que guardemos y carguemos datos, gestionemos la forma en que este proceso puede fallar.

5.2.4 Manejar diferentes tipos de cuentas

El código anterior funcionará correctamente si todo lo que queremos guardar y cargar son cuentas de un tipo particular, ya que todo está escrito en términos de la clase `CuentaCliente`. Sin embargo, nosotros sabemos que nuestro cliente necesita que el programa pueda manejar muchos tipos diferentes de cuenta.

Advertencia de Salud

Este es un apartado complejo. No espero que entienda esto al leerlo a la primera. La razón por la que esta información está expuesta aquí es para complementar. Habiéndole contado todo sobre las jerarquías de clase, y lo útil que es poder basar una nueva clase en una existente (como lo hicimos con `CuentaInfantil`), sería muy injusto por mi parte dejarle descubrir por sí mismo cómo se pueden almacenar y recuperar estos elementos. Este material se proporciona para darle una idea de cómo usted podría realmente crear el funcionamiento de un banco, pero se asume que ya tiene un amplio conocimiento sobre las jerarquías de clases, de la anulación / sobrecarga de métodos y del encadenamiento de constructores.

La buena noticia es que cuando comprenda esto, podrá considerarse realmente por derecho propio un programador C#.

Bancos y Flexibilidad

Nosotros sabemos que cuando nuestro sistema sea utilizado realmente en un banco, habrá una variedad de tipos de cuenta diferentes, algunas de las cuales estarán basadas en otras. Como ejemplo, nosotros hemos examinado/hablado/analizado previamente la `CuentaInfantil`, una cuenta especial para jóvenes que limita la cantidad que puede retirarse a no más de 10 libras. El cliente también nos ha solicitado que la clase `CuentaInfantil`, contenga el nombre del titular de la cuenta "padre". Podríamos implementar dicho comportamiento creando una cuenta que extienda la clase `CuentaCliente`, y añada los comportamientos y propiedades requeridos:

```
public class CuentaInfantil : CuentaCliente
{
    private string nombrePadre;

    public string ObtenerNombrePadre()
    {
       return nombrePadre;
    }

    public override bool RetirarEfectivo( decimal cantidad )
    {
       if ( cantidad > 10 )
       {
           return false;
       }
       return base.RetirarEfectivo(cantidad);
    }

    public CuentaInfantil(
    {
       string nuevoNombre,
       decimal saldoInicial,
       string nombrePadreEn)
       : base(nuevoNombre, saldoInicial)
    }
       nombrePadre = nombrePadreEn;
}
```

Esta es una implementación completa de la `CuentaInfantil`. Esta clase contiene una propiedad adicional, `nombrePadre`, y también se encarga de anular/invalidar el método `RetirarEfectivo` proporcionando el nuevo comportamiento. Tenga en cuenta que he creado un método constructor que se suministra con el nombre del titular de la cuenta, el saldo inicial y el nombre del padre. En este caso, se hace uso del método constructor del padre, para establecer el saldo y el nombre, y a continuación, establece el nombre del padre. Puedo crear una `CuentaInfantil` de la siguiente manera:

```
CuentaInfantil babyJane = new CuentaInfantil("Jane", 20, "John");
```

Esta sentencia crearía una nueva instancia de `CuentaInfantil`, y establecería la referencia babyJane para hacer referencia a ella.

Guardar una clase hija

Cuando queremos guardar una `CuentaInfantil`, también necesitamos guardar la información de la clase padre. Esto resulta ser muy sencillo, siempre y cuando utilicemos el método de guardado que envía la información a una secuencia stream:

```
public override void Guardar(System.IO.TextWriter textoDeSalida)
{
   base.Guardar(textoDeSalida);
   textOut.WriteLine(nombrePadre);
}
```

Este método anula/invalida el método `Guardar` en la `CuentaCliente` padre. Sin embargo, éste primero llama al método reemplazado en el padre para guardar los datos de la `CuentaCliente`. A continuación, realiza el comportamiento de guardado requerido por la clase `CuentaInfantil`. Este es un diseño muy bueno, ya que significa que, si el contenido de datos y el comportamiento de guardado de la clase padre cambian, no es necesario que modifiquemos el comportamiento de la clase hija.

Cargar una clase hija

Podríamos crear un método de carga `static` para la `CuentaInfantil` que lea la información de entrada y construya una nueva `CuentaInfantil`:

```
public static CuentaInfantil Cargar(System.IO.TextReader textoEn)
{
   CuentaInfantil resultado = null;

   try
   {
       string nombre = textoEn.ReadLine();
       string textoSaldo = textoEn.ReadLine();
       decimal saldo = decimal.Parse(textoSaldo);
       string padre = textoEn.ReadLine();
       resultado = new CuentaCliente(nombre,saldo);
   }
   catch
   {
       return null;
   }
   return resultado;
}
```

Sin embargo, no estoy particularmente interesado en este enfoque. Este método rompe una de las reglas/de los principios del buen diseño, ya que ahora nosotros tenemos una *dependencia* entre las dos clases, lo que significa que cuando yo modifico la clase `CuentaCliente` (quizás para agregar un nuevo campo llamado "`pinCliente`"), también tengo que actualizar el método `Cargar` en la clase `CuentaInfantil`, para asegurarme de que estos datos adicionales sean cargados. Si me olvido de hacer esto, el programa compilará, pero cuando se ejecute no funcionará correctamente.

Lo que realmente queremos hacer es que la clase `CuentaCliente` sea responsable de la carga de sus datos, y la clase `CuentaInfantil` solamente se encargue de su contenido; de manera similar a la que el método de guardado utiliza la palabra reservada `base` para guardar el objeto.

La manera de hacerlo es regresando al proceso de construcción. Sabemos que un constructor es un método que toma el control cuando se crea una instancia de una clase. Un constructor puede ser utilizado para establecer los datos en una instancia, y puede recibir la información que le permita hacerlo. En el código anterior existe un constructor para la clase `CuentaInfantil`, que acepta los tres elementos de datos que la `CuentaInfantil` almacena, y a continuación, establece la instancia con estos valores.

Lo que nosotros hacemos es crear constructores para las clases `CuentaCliente` y `CuentaInfantail` que leen la información que necesitan desde una secuencia que se les proporciona:

```
public CuentaCliente Cargar(System.IO.TextReader textoEn)
{
   nombre = textoEn.ReadLine();
   string textoSaldo = textoEn.ReadLine();
   saldo = decimal.Parse(textoSaldo);
}
```

Este constructor establece la nueva instancia `CuentaCliente` leyendo los valores desde la secuencia suministrada. Por lo tanto, puedo escribir un código como este:

```
System.IO.TextWriter textoEn =
          new System.IO.StreamReader(nombredearchivo);
resultado = new CuentaCliente(textoEn);
textoEn.Close();
```

Este código crea una nueva `CuentaCliente` utilizando los datos suministrados desde la secuencia. Ahora, puedo crear un constructor para la `CuentaInfantil` que utilice el constructor de la clase padre:

```
public CuentaInfantil(System.IO.TextReader textoEn) : base (textoEn)
{
    nombrePadre = textoEn.ReadLine();
}
```

Esto elimina la relación de dependencia por completo. Si el comportamiento del constructor de la `CuentaCliente` cambia, nosotros no tenemos que cambiar para nada la `CuentaInfantil`.

Tenga en cuenta que estos constructores no realizan ninguna comprobación de errores, solo lanzan excepciones si algo sale mal. Teniendo en cuenta de todos modos, que la única forma en que un constructor puede fallar es lanzando una excepción, este es un comportamiento razonable.

Interfaces y la operación de guardado

Cuando comenzamos el desarrollo de esta cuenta, establecimos una interfaz que describía todos los comportamientos que una instancia de cuenta debería poder hacer. Al principio, ésta no incluía los comportamientos de guardado, pero ahora podríamos actualizarla para incluirlos:

```
public interface ICuenta
{
   void IngresarEfectivo( decimal cantidad );
   bool RetirarEfectivo( decimal cantidad );
   decimal ObtenerSaldo();
   string ObtenerNombre();

   bool Guardar(string nombredearchivo);
   void Guardar(System.IO.TextWriter textoDeSalida);
}
```

Ahora podemos solicitar cualquier elemento que implemente la interfaz `ICuenta` para que se guarde en un archivo o una secuencia. Esto resulta muy útil cuando nosotros almacenamos nuestra colección de cuentas, ya que el contenedor de la cuenta no tendrá que comportarse de manera diferente dependiendo del tipo de cuenta que esté guardando, solamente tiene que llamar al método de guardado para la instancia particular. Esto es una buena idea.

Carga y fábricas

Cuando se trata de nuevo de cargar nuestras clases, las cosas se complican un poco más. Podría pensar que sería sensato añadir métodos de carga a la interfaz `ICuenta`, para que podamos solicitar que se carguen instancias de un tipo particular de cuenta. Sin embargo, en este caso nos encontramos con un problema, ya que cuando cargamos las cuentas, nosotros realmente no tenemos una instancia para llamar a alguno de los métodos. Además, teniendo en cuenta que estamos leyendo de una secuencia de datos, en realidad no sabemos qué tipo de elemento estamos cargando.

La solución a este problema es identificar el tipo de cada instancia en la secuencia cuando nosotros guardamos las clases. El método de guardado para nuestro banco podría tener este aspecto:

```
public void Guardar(System.IO.TextWriter textoDeSalida)
{
   textoDeSalida.WriteLine(tablaHashBancaria.Count);
   foreach (CuentaCliente cuenta in tablaHashBancaria.Values)
   {
       textoDeSalida.WriteLine(cuenta.ObtenerTipo().Nombre);
       cuenta.Guardar(textoDeSalida);
   }
}
```

Este código se parece mucho a nuestro método original, exceptuando la línea adicional, que ha sido resaltada. Este método escribe el nombre de la clase, y hace uso del método `ObtenerTipo`(), que puede ser invocado desde una instancia de una clase para obtener el tipo de esa clase. Tras conseguir el tipo, nosotros también podemos obtener la propiedad `Nombre` de este tipo e imprimirla. En otras palabras, si la cuenta es de tipo `CuentaCliente`, el programa imprimirá:

```
CuentaCliente
Rob
100
```

El resultado ahora contiene el nombre del tipo de cada clase que se ha escrito. Lo que significa que cuando se vuelva a leer la secuencia, se puede utilizar esta información para crear el tipo correcto de clase. La forma más adecuada de hacerlo es creando una *fábrica (un patron de fabricación)*, que producirá instancias de la clase requerida:

```
class FabricaDeCuentas
{
    public static ICuenta CrearCuenta(
        string nombre, System.IO.TextReader textoEn)
    {
        switch (nombre)
        {
           case "CuentaCliente":
               return new CuentaCliente(textoEn);

           case "CuentaInfantil":
               return new CuentaInfantil(textoEn);

           default:
               return null;
        }
    }
}
```

Esta clase solamente contiene un método declarado con el modificador `static`. Este método espera que se le pasen dos parámetros, el nombre de la clase que se creará y una secuencia de lectura. Este utiliza el nombre para decidir qué elemento crear, lo crea, y a continuación, lo devuelve al llamador. Si el nombre no se reconoce, devuelve `null`. El método de carga del banco, puede utilizar esta fábrica para crear instancias de cuentas a medida que se cargan:

```
public static HashBancario Cargar(System.IO.TextReader textoEn)
{
   HashBancario resultado = new HashBancario();
   string contarCadena = textoEn.ReadLine();
   int contador = int.Parse(contarCadena);
```

```
        for (int i = 0; i < count; i++)
        {
            string nombreClase = textoEn.ReadLine();
            ICuenta cuenta = FabricaDeCuentas.CrearCuenta(nombreClase, textoEn);
            resultado.tablaHashBancaria.Add(cuenta.ObtenerNombre(), cuenta);
        }
            return resultado;
    }
```

Una vez más, este código se parece mucho a nuestro método de carga original, exceptuando que en este método se utiliza una fábrica para crear instancias de cuenta, una vez que éste ha leído el nombre de la clase desde la secuencia.

```
class ProgramaBancario
{
    public static void Main()
    {
        DictionaryBancario nuestroBanco = new DictionaryBancario();

        CuentaCliente nuevaCuenta = new CuentaCliente("Rob", 1000000);

        if (nuestroBanco.AlmacenarCuenta(nuevaCuenta))
            Console.WriteLine("CuentaCliente añadida al banco");

        CuentaInfantil nuevaCuentaInfantil =
            new CuentaInfantil("David", 100, "Rob");

        if (nuestroBanco.AlmacenarCuenta(nuevaCuentaInfantil))
            Console.WriteLine("CuentaInfantil añadida al banco");

        nuestroBanco.Guardar("Prueba.txt");

        DictionaryBancario cargarBanco =
            DictionaryBancario.Cargar("Prueba.txt");

        ICuenta cuentaAlmacenada = cargarBanco.EncontrarCuenta("Rob");
        if (cuentaAlmacenada != null)
            Console.WriteLine("CuentaCliente encontrada en el banco");
        cuentaAlmacenada = cargarBanco.EncontrarCuenta("David");
        if (cuentaAlmacenada != null)
            Console.WriteLine("CuentaInfantil encontrada en el banco");
    }
}
```

Código de Ejemplo 51 Clase Fábrica de Cuentas en acción

Este código muestra cómo testeo mi fábrica. Este crea un banco, añade dos cuentas diferentes y a continuación, las vuelve a cargar.

Dependencias de Fábrica

Tenga en cuenta que ahora tenemos una verdadera dependencia, entre nuestro sistema y la clase de fábrica. Si alguna vez añadimos un nuevo tipo de cuenta, necesitaremos actualizar el comportamiento de la fábrica, para que contenga una expresión de coincidencia para tratar con el nuevo tipo de cuenta. Hay, de hecho, una forma de eliminar la necesidad de hacerlo. Esta implica escribir código que buscará clases C# que implementen interfaces particulares y las creen automáticamente. No obstante, este tipo de cosas, van más allá del propósito de este texto.

NOTAS BANCARIAS: CÓDIGO DESORGANIZADO

Podría pensar que las soluciones anteriores tienen un código algo desorganizado. La forma en que gestionamos el guardado de elementos (mediante el uso de un método en la clase), y la carga (mediante el uso de un constructor), no es simétrica. Sin embargo, no hay nada malo en esta forma de trabajar. Cuando quiera realizar movimientos de objetos a almacenamiento y a la inversa, se encontrará antes estos problemas, y esta solución es tan buena como cualquier otra que yo hubiese encontrado.

5.3 Objetos de Negocio y Edición

Nosotros hemos visto cómo diseñar una clase que puede ser utilizada para almacenar la información de los clientes de nuestro banco. Ahora sabemos cómo guardar la información en una instancia de clase, y también a realizar esto para un gran número de elementos de una variedad de tipos diferentes. Ahora nosotros necesitamos considerar cómo podemos hacer que nuestro sistema de administración de cuentas sea realmente útil, proporcionando una interfaz de usuario que permita a las personas interactuar con nuestros objetos de negocio que se ejecutan en el banco.

Punto del Programador: Código en fase de producción

A partir de ahora, todos los códigos de ejemplo van a ser proporcionados en un contexto de *código en fase de producción*. Este es el tipo de código de un producto que estaría orgulloso de proporcionar en su versión final. Lo que significa que voy a considerar qué aspectos debería analizar cuando se encuentra escribiendo código para clientes reales. Esto puede hacer que los ejemplos sean un poco más complejos de lo que a usted le gustaría que fueran, pero creo que vale la pena que sea así, ya que empezará a comprender el código que está escribiendo, y los problemas cono lo que está lidiando, que son con los que los programadores "reales" se enfrentan en el día a día.

5.3.1 El rol de los Objetos de Negocio

Hemos tomado una línea muy estricta en nuestro sistema bancario evitando que cualquiera de los componentes de la cuenta bancaria se comunique realmente con el usuario. Esto es a causa de que clases como `CuenteCliente` son lo que se denomina *objetos de negocio*. No forma parte del trabajo de estas comunicarse con los usuarios; sino tratar estrictamente de realizar un seguimiento de la información existente en la cuenta bancaria del cliente. El cometido del objeto de negocio es asegurarse de que los datos que éstas contienen siempre sean los adecuados. Los comportamientos de modificación deben hacer uso de los métodos que expone el objeto de negocio.

Gestionar un nombre de cuenta bancaria

Como ejemplo, considere el nombre del titular de la cuenta bancaria. Es importante que éste siempre se almacene de forma segura, pero la gente en ocasiones puede querer cambiar su nombre. Esto significa que el objeto de negocio debe proporcionar una manera para que el nombre pueda ser modificado.

Sin embargo, este es un proceso que va más allá de simplemente cambiar una cadena por otra. El nuevo nombre debe ser válido. Lo que significa que la cuenta debe poder rechazar nombres que no sean adecuados. Si se va a rechazar los nombres sería muy útil para el usuario si se les pudiera

comunicar la razón de por qué un nombre determinado no es válido, por lo que el proceso de validación debe proporcionar retroalimentación sobre por qué no se aceptó y se produjo el error.

Un buen ingeniero de software proporcionaría un código como este:

```
private string nombre;

public string ObtenerNombre()
{
   return this.nombre;
}

public static string ValidarNombre(string nombre)
{
   if (nombre == null) {
       return "Parámetro Nombre null";
   }
   string nombreDeEspaciosEnBlanco = nombre.Trim();
   if (nombreDeEspaciosEnBlanco.Length == 0)
   {
       return "El nombre introducido no contiene texto";
   }
   return "";
}

public bool EstablecerNombre(string nombreEn)
{
   string respuesta;
   respuesta = ValidarNombre(nombreEn);
   if (respuesta.Length > 0)
   {
       return false;
   }
   this.nombre = nombreEn.Trim();
   return true;
}
```

El nombre es almacenado como un miembro `private` de la cuenta clase. El programador ha proporcionado tres métodos que permiten a los usuarios de mi clase `Cuenta` lidiar con los nombres. Hay un método para **obtener** el nombre, otro para **establecerlo**, y otro para **validarlo**. El método de validación es invocado por el método de establecimiento para que mi objeto de negocio se asegure de que una cuenta nunca contenga un nombre no válido. El método de validación rechazará un nombre de cadena que esté vacío o simplemente contenga espacios en blanco. Éste recorta todos los espacios antes y después del texto del nombre y, a continuación, verifica si la cadena resultante está vacía. Si es así, la cadena es rechazada.

Yo proporciono el método de validación para que los usuarios de mi clase puedan comprobar sus nombres y asegurarse de que estos sean válidos. Esto hace que no me vea involucrado en mucho trabajo adicional, ya que también utilizo el propio método a la hora de validar el nombre. El método de validación devuelve una cadena que proporciona un mensaje de error si la cadena es rechazada. Puede haber varias razones por las cuales un nombre no es válido, en este momento acabo de dar dos de ellas. Las razones por las cuales un nombre no es válido son, por supuesto, parte de los *metadatos* del proyecto. Yo también, he creado este método `static`, para que los nombres puedan ser validados sin necesidad de tener una instancia real de la cuenta.

Testear el establecimiento de Nombres

Por supuesto, una vez que tenemos establecidos estos métodos, lo más natural es realizar pruebas para testearlos:

```
int contadorDeErrores = 0;
string respuesta;
respuesta = CuentaCliente.ValidarNombre(null);
if (respuesta != "Parámetro Nombre null")
{
   Console.WriteLine("Prueba de Nombre null fallida");
   contadorDeErrores++;
}
respuesta = CuentaCliente.ValidarNombre("");
if (respuesta != "El nombre introducido no contiene texto")
{
   Console.WriteLine("Prueba de Nombre vacío fallida");
   contadorDeErrores++;
}
respuesta = CuentaCliente.ValidarNombre("  ");
if (respuesta != "El nombre introducido no contiene texto")
{
   Console.WriteLine("Prueba de Nombre con cadena en blanco fallida");
   contadorDeErrores++;
}
CuentaCliente a = new CuentaCliente("Rob", 50);
if (!a.EstablecerNombre("Jim"))
{
   Console.WriteLine("EstablecerNombre Jim fallido);
   contadorDeErrores++;
}
if (a.ObtenerNombre() != "Jim")
{
   Console.WriteLine("ObtenerNombre Jim fallido");
   contadorDeErrores++;
}
if (!a.EstablecerNombre("   Pete   "))
{
   Console.WriteLine("EstablecerNombre Pete Espacios en blanco eliminados
                      fallido");
   contadorDeErrores++;
}
if (a.ObtenerNombre() != "Pete")
{
   Console.WriteLine("ObtenerNombre Pete fallido");
   contadorDeErrores++;
}
if (contadorDeErrores > 0 )
{
   HacerSonarSirena();
}
```

Código de Ejemplo 52 Testear el establecimiento de Nombres

Estas son todas las pruebas que se me han ocurrido. Primeramente, testeo el método `ValidarNombre` para asegurarme de que rechaza ambos tipos de cadena vacía. A continuación, me aseguro de poder establecer el nombre de la cuenta. Finalmente, compruebo que el método encargado de la eliminación de espacios en blanco para los nombres de cadena funciona correctamente.

Punto del Programador: Utilice Números, no Mensajes

Hay un problema que no he abordado en mis programas de ejemplo, que les impide ser completamente perfectos. Y no es otro que el del manejo de errores. Por el momento, los errores proporcionados por mis métodos de validación son cadenas de texto. En un entorno de producción genuino, los errores serían valores numéricos. Esto se debe a que los valores numéricos son mucho más fáciles de identificar en el código de las pruebas, y también porque haría que mi programa pudiera funcionar en un idioma foráneo muy fácilmente. Todo lo que necesitaría es una tabla de búsqueda para convertir el número de mensaje a una cadena apropiada en el idioma activo actual. Debería considerar asuntos como estos como parte de los *metadatos* de su proyecto. El hecho de que el sistema deba funcionar en Francia y Alemania es algo que debe tener en cuenta desde el principio.

Editar el Nombre

Nosotros ahora tenemos un objeto de negocio que proporciona métodos que nos permiten editar el valor del nombre. Ahora puedo escribir el código para modificar esta propiedad:

```
while (true)
{
    Console.Write("Introduzca un nuevo nombre: ");
    nuevoNombre = Console.ReadLine();
    string respuesta;
    respuesta = cuenta.ValidarNombre(nuevoNombre);

    if (respuesta.Length == 0)
    {
       break;
    }
    Console.Write("Nombre no válido: " + respuesta);
}

cuenta.EstablecerNombre(nuevoNombre);
```

Este código modificará el nombre para una instancia de cuenta a la que se hace referencia mediante `cuenta`. Se leerá un nuevo nombre, y si el nombre es válido, se abandonará el bucle y el nombre será establecido. Si el nombre no es válido, se imprimirá el mensaje que indica esto por pantalla y el bucle se repetirá. Esta no es la solución más elegante, pero funciona correctamente y mantiene

al usuario informado de lo que está ocurriendo. Ahora que nosotros tenemos nuestro código de modificación tenemos que incluirlo en alguna parte del código.

Crear una clase Editora

La mejor manera de hacerlo es crear una clase que tenga el cometido de realizar la edición. Esta clase realizará el trabajo sobre una cuenta particular que necesite ser editada. Existirá una dependencia entre la clase editora y la clase cuenta, ya que si la clase cuenta cambia la clase editora necesita ser actualizada, pero esto es algo con lo que tendremos que convivir.

Cuando quiero modificar una cuenta, creo una instancia de la clase editora, y le paso una referencia a la instancia de la cuenta:

```
public class IUModificarTextoDeCuenta
{
    private ICuenta cuenta;

    public IUModificarTextoDeCuenta(Cuenta cuentaEn)
    {
       this.cuenta = cuentaEn;
    }

    public void ModificarNombre()
    {
       string nuevoNombre;
       Console.WriteLine("Modificar nombre");

       while (true)
       {
          Console.Write("Introduzca un nuevo nombre: ");
          nuevoNombre = Console.ReadLine();
          string respuesta;
          respuesta = this.cuenta.ValidarNombre(nuevoNombre);

          if (respuesta.Length == 0)
          {
             break;
          }
          Console.WriteLine("Nombre no válido: " + respuesta);
       }
       this.cuenta.EstablecerNombre(nuevoNombre);
    }
}
```

Este es mi clase editora de cuentas. Por el momento, solo puede editar el nombre, pero añadiré otros métodos de edición más adelante. La clase mantiene un seguimiento de la cuenta que está

modificando, por lo que le paso una referencia a esta cuenta cuando la construyo. Yo utilizaría el editor de nombre de la siguiente manera:

```
CuentaCliente a = new CuentaCliente("Rob", 50);
IUModificarTextoDeCuenta modificar = new IUModificarTextoDeCuenta(a);
modificar.ModificarNombre();
```

Código de Ejemplo 53 Editar el Nombre

Este código crea una instancia de una cuenta cliente. A continuación, crea un objeto editor y le pide que edite el nombre de esa cuenta.

Tenga en cuenta que a la clase editora se le pasa una referencia a la interfaz `ICuenta`, no una referencia a la clase cuenta. Esto significa que cualquier cosa que se comporte como una cuenta puede ser editada utilizando esta clase. Tenga en cuenta también que el editor recuerda la clase de cuenta que se está editando para que cuando llame al método `ModificarNombre` pueda trabajar sobre esa referencia.

Punto del Programador: Acostúmbrese a pasar referencias entre métodos

Es importante que se acostumbre a la idea de pasar referencias entre métodos. Si yo quiero "darle" algún elemento a la clase para que trabaje con él, lo haré invocando a un método en esa clase y pasándole la referencia a dicho elemento como parámetro. Usted utilizará esta técnica a la hora de crear diferentes formularios de edición.

5.3.2 Un Sistema de Edición Basado en Texto

Ahora tenemos un único método en nuestra clase editora, que puede ser utilizada para modificar el nombre de una cuenta bancaria. Lo que realmente queremos es un sistema de menú que reciba un comando por parte del usuario y realice esa función:

```
Editar cuenta de Pete
    Introducir nombre para modificar el nombre
    Introducir ingresar para ingresar efectivo
    Introducir retirar para retirar efectivo
    Introducir salir para salir del programa
Introducir comando:
```

El usuario teclea el nombre del comando requerido y, a continuación, el programa ejecuta esa función.

Mi método de edición lee repetidamente los comandos y los envía al método apropiado. Yo he extendido la clase cuenta para gestionar el saldo de la cuenta, y he añadido métodos de edición para el ingreso y la retirada de efectivo. También debe tener en cuenta que yo elimino los espacios en blanco de la cadena y la convierto en minúsculas antes de utilizarlas para dirigirla a la construcción `switch` seleccionada por el comando.

El método de edición pasa una referencia a la cuenta que está siendo editada. Ésta, a continuación, pasa dicha referencia a los métodos de servicio que realizan realmente el trabajo.

```
      public void RealizarEdición(CuentaCliente cuenta)
      {
      string comando;
      do
      {
         Console.WriteLine("Editar cuenta para {0}",
             cuenta.ObtenerNombre() );
         Console.WriteLine("Introducir nombre para modificar el nombre");
         Console.WriteLine("Introducir ingresar para ingresar efectivo");
         Console.WriteLine("Introducir retirar para retirar efectivo");
         Console.WriteLine("Introducir salir para salir del programa");
         Console.Write("Introducir comando: ");
         comando = Console.ReadLine();
         comando = comando.Trim();
         comando = comando.ToLower();

         switch ( comando )
         {
            case "nombre" :
               ModificarNombre(cuenta);
               break;
            case "ingresar" :
               IngresarEfectivo(cuenta);
               break;
            case " retirar" :
               RetirarEfectivo(cuenta);
               break;
         }
      } while ( comando != "salir" );
   }
```

Punto del Programador: Cada Mensaje cuenta

Debe recordar que cada vez que su programa muestra texto al usuario podría tener un problema causado por el idioma utilizado. El método de ejemplo que he escrito para mostrar el menú de edición de mi cuenta bancaria, imprime por pantalla las cadenas de texto que he integrado en el propio código en el idioma español. En un programa desarrollado correctamente, este problema sería gestionado por la identificación numérica de los mensajes, haciendo mucho más sencillo el cambio de idioma de las cadenas de texto que se muestran al usuario por pantalla.

Como regla general en un sistema de producción, nunca debe escribir directamente en los métodos las cadenas de texto que se presentan al usuario por pantalla (esto incluye los nombres de los comandos que el usuario puede introducir). Todo debe ser manejado en términos de números de mensaje. La única excepción a esta regla, es la situación en la que el cliente le ha asegurado que el programa solo debe mostrarse en un idioma particular. Las bibliotecas de C# contienen un conjunto de recursos que le ayudan a gestionar la internacionalización de sus programas.

NOTAS BANCARIAS: MÁS DE UNA INTERFAZ DE USUARIO

El banco puede contar con un amplio abanico de servicios desde los que se permitan modificar los detalles de la cuenta. Estos servicios pueden ser utilizados por un operador que trabaje en un centro de llamadas (call center), un operador que atienda un terminal de punto de venta, un cliente desde su propio teléfono móvil o un cliente desde un cajero automático. Espero que pueda comprender que la única manera de gestionar todas estas diferentes formas de interactuar con la cuenta bancaria es separando el objeto de negocio (la propia cuenta en sí) del comportamiento de entrada/salida utilizado.

Tenga en cuenta que nosotros **nunca** debemos permitir que sea el código de la interfaz de usuario el qué constituya un nombre como válido. Este asunto no es responsabilidad del front end. El único elemento que puede dar validez a un nombre es el propio objeto de cuenta en sí. La interfaz de usuario siempre debe trabajar sobre la base que utilizará el objeto de negocio para realizar este tipo de validación.

5.4 Threads y Threading - Programación con Hilos (Procesos y Subprocesos)

Si quiere considerarse programador, necesita conocer algunos aspectos relacionados con los Hilos (Threads). Estos pueden hacer que los programas sean mucho más fáciles de crear, pero también pueden ser una fuente de errores realmente molestos. En el futuro, donde la velocidad de los

procesadores de las computadoras estará limitada por cosas tan irritantes como la velocidad de la luz y el tamaño de los átomos, ellos serán el medio mediante el cual podremos seguir mejorando el rendimiento de nuestros sistemas informáticos.

5.4.1 ¿Qué es un Hilo?

Por ahora, nuestros programas solamente utilizan un *hilo* cuando se ejecutan. El hilo generalmente comienza en el método `Main` y termina cuando se alcanza el final de este método. Puede pensar en un hilo como un tren que se desplaza a lo largo de un trayecto de vía. La vía la forman las distintas instrucciones que el hilo está ejecutando. De la misma manera que se puede poner más de un tren en una sola vía, una computadora puede ejecutar más de un hilo en un único bloque de código de programa.

Una computadora puede soportar múltiples hilos de dos maneras. La primera de ellas es conmutando rápidamente entre los hilos activos, dando a cada hilo la oportunidad de ejecutarse durante un tiempo limitado antes de pasar a ejecutar otro. Cuando un hilo no se está ejecutando, permanece en estado de "congelación", esperando que le vuelva el turno de ejecución. En lo que al hilo respecta, éste se ejecuta continuamente, pero en realidad un hilo solo puede estar activo durante una pequeña fracción de segundo cada cierto tiempo.

La segunda forma de soportar varios hilos es tener una computadora con más de un procesador. Las primeras computadoras tenían solamente un procesador y, por lo tanto, para soportar varios hilos, se veían forzadas a dividir el tiempo asignado a cada hilo. Las máquinas de hoy en día tienen procesadores de doble núcleo "dual core", o incluso de cuatro núcleos "quad core", por lo que pueden ejecutar varios hilos simultáneamente.

5.4.2 ¿Por qué tenemos y debemos crear Hilos?

Los hilos hacen posible que las computadoras hagan alguna tarea útil mientras un programa se encuentra detenido a la espera de órdenes. El sistema puede estar reproduciendo música y al mismo tiempo estar descargando archivos mientras otro programa se encuentra a la espera de que se presione una tecla. Una computadora actual puede ejecutar muchos millones de instrucciones por segundo; y, por tanto, tiene sentido compartir esta capacidad entre varias tareas.

Los programas que ha estado escribiendo se han ejecutado como hilos en su computadora. Esta es la forma en que su programa se mantiene activo al mismo tiempo que lo hacen otros programas que desee utilizar junto con él. Usted probablemente no ha sido consciente de esto porque el sistema operativo (por lo general Windows), realiza un buen trabajo para compartir la potencia de procesamiento.

Los hilos proporcionan otro nivel de "abstracción", en el que, si nosotros queremos hacer más de una cosa a la vez en el sentido de que si deseamos hacer más de una cosa al mismo tiempo, es muy útil simplemente mandar a un hilo para realizar la tarea, en lugar de intentar entrelazar la segunda

tarea con la primera. Si nosotros estuviésemos desarrollando un procesador de textos, nos resultaría sumamente útil crear diferentes hilos para realizar las tareas que requieren de más tiempo, como la impresión y la revisión ortográfica. Estas tareas podrían realizarse "en segundo plano" mientras el usuario sigue trabajando en el documento.

Los hilos también son la forma en que los formularios de Windows responden a los eventos. Usted ha visto que se puede hacer un evento "clic" desde un componente Button para ejecutar un método de controlador de eventos. En realidad, el sistema Windows crea un nuevo hilo de ejecución que ejecuta el código del controlador de eventos.

Punto del Programador: Los hilos pueden ser peligrosos

De la misma manera que permitir que dos trenes compartan la misma vía ferroviaria a veces puede ocasionar problemas, los programas que utilizan hilos también pueden fallar de manera espectacular y confusa. Es posible que los sistemas fallen solo cuando una determinada secuencia de eventos hace que dos subprocesos que anteriormente se comportaron bien se "peleen" por un elemento de datos compartido. Los bugs causados por la técnica que permite que una aplicación ejecute simultáneamente varias operaciones en el mismo espacio de proceso "*threading*", se encuentran entre los más difíciles de rastrear y corregir porque hay que lograr que el problema suceda antes de poder corregirlo, y las circunstancias que hacen que se produzca el fallo sólo pueden ocurrir de vez en cuando. Voy a ofrecerle algunos consejos para que se asegure de que sus programas no se vean afectados por errores relacionados con el threading, y le sugiero que los siga cuidadosamente.

5.4.3 Hilos y Procesadores

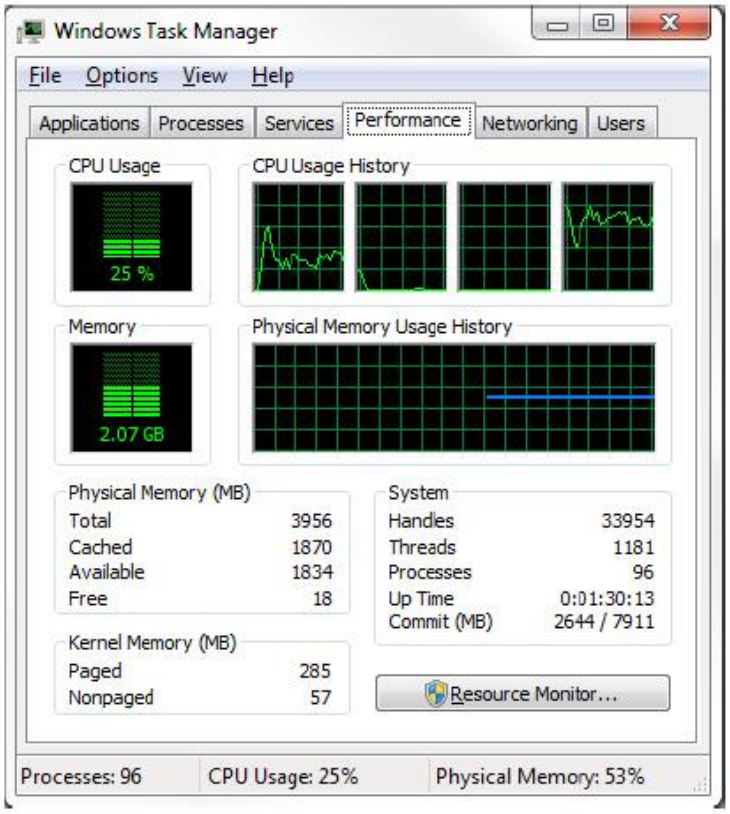
Considere el siguiente método:

```
static private void bucleOcupado()
{
   long contador;
   for (contador = 0; contador < 1000000000000L; contador = contador + 1)
   {
   }
}
```

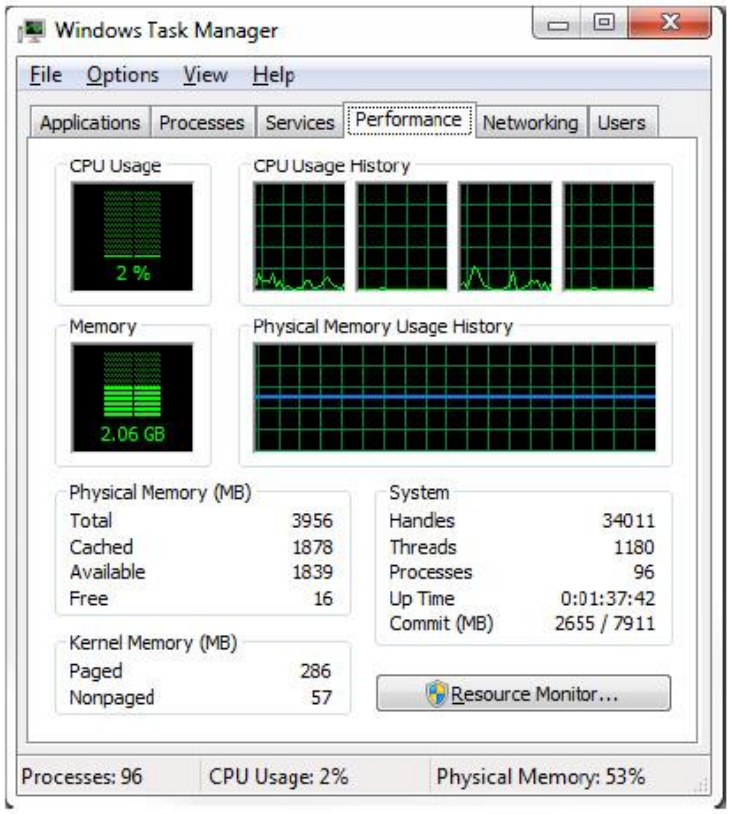
Código de Ejemplo 54 Hilo simple bucle ocupado

Este método no hace nada, pero se ejecuta millones de veces. Si ejecuto un programa que invoca a este método, lo primero que ocurre es que el programa parece detenerse. Tras un breve período de tiempo, el ventilador de mi laptop comienza a girar con mayor rapidez para disipar el aumento

de temperatura del procesador. Si inicio el Administrador de tareas de Windows, vería algo similar a lo que aparece en la siguiente imagen:



Mi laptop tiene un procesador de cuatro núcleos, lo que significa que puede ejecutar cuatro hilos al mismo tiempo. Usted puede ver pequeñas gráficas en la parte del historial de uso de la CPU (Unidad Central de Procesamiento) que muestra cuán ocupado está cada procesador. Por el aspecto de la gráfica anterior, el procesador de la izquierda y el procesador de la derecha están bastante ocupados, pero los dos procesadores del medio no están haciendo nada. Si detengo la ejecución de mi programa, visualizaré en el Administrador de tareas algo similar a lo siguiente.



Ahora todos los procesadores se encuentran en ralentí. Cuando nosotros empecemos a utilizar más de un hilo, deberíamos visualizar un mayor número de gráficas en uso y alcanzando picos más altos, ya que nuestros programas se ejecutarán utilizando múltiples procesadores.

Punto del Programador: Múltiples hilos pueden mejorar el rendimiento

Si puede determinar cómo distribuir su programa en varios hilos, esto puede marcar una gran diferencia en la velocidad con la que se ejecuta. El método anterior solo puede utilizar un máximo del 25% de los recursos de mi sistema, ya que solo se ejecuta en uno de los cuatro procesadores disponibles.

5.4.4 Ejecutar un Hilo

Tenga en cuenta que estas descripciones de comportamiento de los hilos son para el entorno de Microsoft .NET, y no son estrictamente parte del propio lenguaje C#. Si escribe programas para el sistema operativo Windows 8, puede estar creando software basado en Windows RT. En este caso, encontrará que los hilos se utilizan de una manera ligeramente diferente.

Un hilo individual es gestionado/manejado por su programa como una instancia de la clase `Thread`. Esta clase vive en el espacio de nombres `System`. La manera más sencilla de asegurarnos de que podemos utilizar todos los recursos que permiten que una aplicación ejecute simultáneamente varias operaciones "threading", es añadiendo la directiva `using` al inicio del código de nuestro programa:

```
using System.Threading;
```

La clase `Thread` suministra un vínculo entre el programa y el sistema operativo. Uno de los trabajos del sistema operativo es gestionar los hilos en su equipo y decidir exactamente cuándo debe llegar a ejecutarse cada uno de ellos. Utilizando los métodos proporcionados por la clase `Thread`, puede iniciarlos y detenerlos.

Seleccionar dónde comienza a ejecutarse un Hilo

Cuando crea un hilo, necesita indicar al hilo en qué punto debe iniciar su ejecución. Esto es como decirle a su hermano más pequeño en qué parte de la pista le gustaría colocar su tren. Puede hacer esto mediante el uso de *delegados*. Un delegado es una forma de referirse a un método en una clase. Cuando se crea un hilo administrado, el método que se ejecuta en el hilo queda representado por un delegado `ThreadStart`. Este tipo de delegado puede hacer referencia a un método que no devuelve un valor ni acepta ningún parámetro. Si echa un vistazo al método `bucleOcupado` que creamos anteriormente, comprobará que este método se ajusta perfectamente a la norma. Puede crear un delegado `ThreadStart` para referirse a este método de la siguiente manera:

```
ThreadStart metodoBucleOcupado = new ThreadStart(bucleOcupado);
```

Crear un Hilo

Una vez que ya ha definido el punto de inicio, ahora puede crear un valor `Thread`:

```
Thread t1 = new Thread(metodoBucleOcupado);
```

La variable `t1` ahora hace referencia a una instancia del hilo. Tenga en cuenta que en este momento el hilo no se está ejecutando, sino que se encuentra a la espera de ser iniciado. Cuando nosotros comencemos a ejecutar `t1`, éste sigue las instrucciones del `metodoBucleOcupado`, y hace una llamada al método `bucleOcupado`.

Iniciar un Hilo

La clase `Thread` proporciona una cantidad de métodos que su programa puede usar para controlar lo que éste hace. Para iniciar la ejecución del hilo puede utilizar el método `Start`:

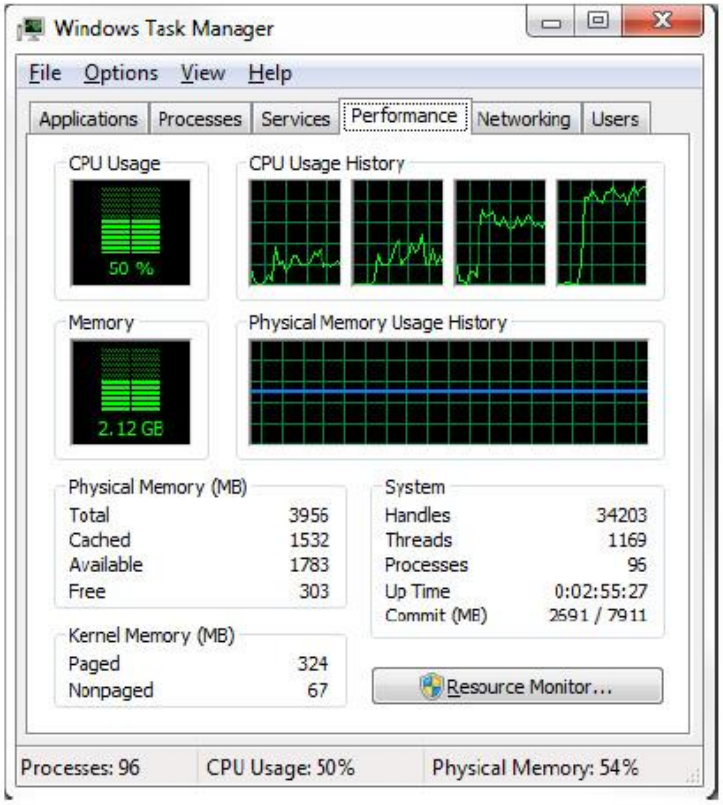
```
t1.Start();
```

Este es el punto en el que el hilo comienza a ejecutarse.

```
class DemoSubproceso
{
    static private void bucleOcupado()
    {
        long contador;

        for (contador = 0; contador < 1000000000000L; contador=contador+1)
        {
        }
    }

    static void Main()
    {
        ThreadStart metodoBucleOcupado = new ThreadStart(bucleOcupado);

        Thread t1 = new Thread(metodoBucleOcupado);

        t1.Start();
        bucleOcupado();
    }
}
```

Código de Ejemplo 55 Dos Hilos bucle ocupado

Este código crea un hilo llamado `t1`, que inicia y ejecuta el método `bucleOcupado`. Este también llama directamente al `bucleOcupado` desde el método `Main`. Esto significa que hay dos procesos activos. Esto cambia el resultado que veíamos anteriormente desde el Administrador de tareas:



Ahora todas las gráficas se visualizan en uso mostrando actividad, y el uso de CPU ha aumentado del 25% al 50%, lo que demuestra que nuestro programa está utilizando más procesadores.

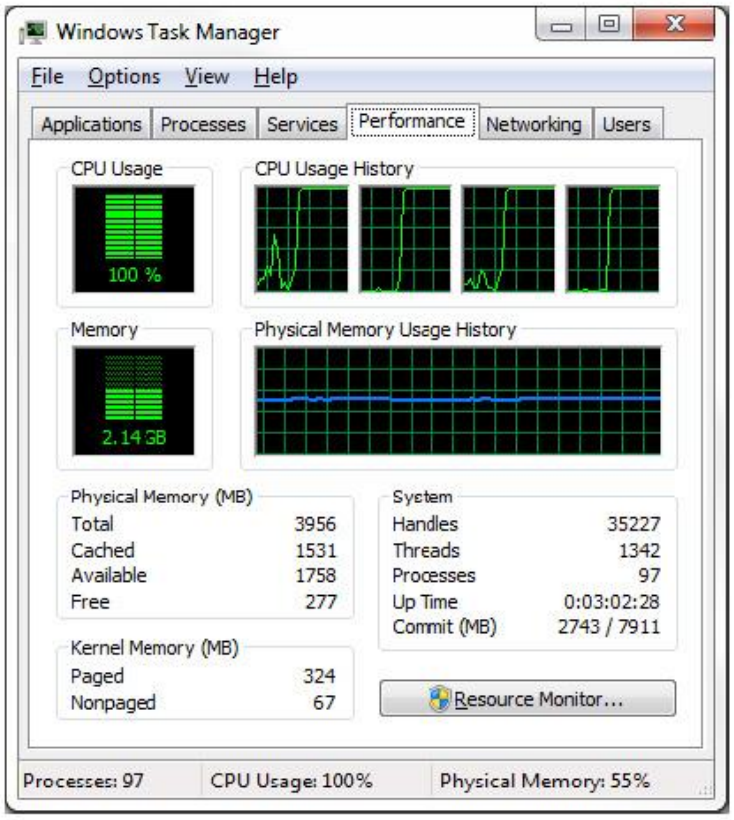
Crear un mayor número de Hilos

En la imagen anterior puede ver que, si creamos un hilo adicional, podemos hacer un uso más exhaustivo de la potencia que nos ofrece la computadora. De hecho, nosotros podemos crear más hilos como este:

```
for (int i = 0; i < 100; i = i + 1)
{
   Thread t1 = new Thread(metodoBucleOcupado);
   t1.Start();
}
```

Código de Ejemplo 56 Cien Hilos bucle ocupado

Este bucle creará 100 hilos, todos ejecutando el método bucleOcupado. Ahora nuestra computadora está realmente ocupada:



Todos los procesadores se encuentran ahora trabajando al máximo, y el uso de la CPU ha aumentado hasta alcanzar el 100%. Si ejecuta este código, comprobará que la máquina tarda más en responder, y que los ventiladores se encuentran trabajando al máximo rendimiento.

Punto del Programador: Demasiados Hilos harán que todo se ralentice

Tenga en cuenta que nada ha impedido que el programa se inicie con un gran número de hilos en ejecución. Esto es potencialmente peligroso. Si se ejecutan demasiados hilos a la vez se ralentizará todo. En el campo de la seguridad informática, esta es la base utilizada por un ataque del tipo de *Denegación de Servicio*. Este hace que su computadora deje de responder, al haberse iniciado una gran cantidad de hilos que sobrecargan los recursos que se asignan al procesador. Afortunadamente, Windows da mayor prioridad a los hilos que se ocupan de los dispositivos de entrada, por lo que normalmente puede tomar el control y detener ese comportamiento errático.

5.4.5 Hilos y Sincronización

Es posible que se pregunte por qué todos los hilos no pelean por el valor de la variable `contador` que está siendo utilizada para recorrer el bucle. Esto se debe a que la variable es *local* al método `bucleOcupado`:

```
static private void bucleOcupado()
{
    long contador;

    for (contador = 0; contador < 1000000000000L; contador = contador+1)
    {
    }
}
```

La variable `contador` ha sido declarada dentro del cuerpo del método. Esto significa que cada método tiene su propia copia de esta variable *local*. Si usted todavía tiene en mente la idea de que los hilos son como trenes, puede mantenerla para este caso, y considerar que los datos de cada hilo se encuentran contenidos en vagones que arrastrados por el tren. Nosotros podríamos hacer un pequeño cambio en este código, y convertir a la variable `contador` es un miembro de la clase:

```
static long contador;
static private void bucleOcupado()
{
    for (contador = 0; contador < 1000000000000L; contador = contador+1)
    {
    }
}
```

Este método presenta un aspecto muy similar al anterior, pero ahora cada hilo que ejecuta el método `bucleOcupado` comparte la misma variable `contador`. Esto es potencialmente catastrófico. Ahora es muy difícil predecir cuánto tiempo llevará a este programa multi-hilo finalizar el recorrido del bucle. Considere la siguiente secuencia de acciones:

1. El Hilo 1 obtiene el valor de la variable `contador` para que éste pueda sumarle 1.

2. Antes de que la suma pueda ser realizada, el Hilo 1 es detenido, y se autoriza al Hilo 20 que se inicie y se ejecute.

3. El Hilo 20 obtiene el valor de la variable `contador`, le suma uno a ésta y la guarda de nuevo en memoria.

4. Pasado un tiempo, el Hilo 1 obtiene el control de nuevo, suma 1 al valor de la variable que obtuvo anteriormente, se detiene, y almacena el resultado en la memoria**.**

Debido a la forma en la que el Hilo 1, fue interrumpido durante su cálculo, éste sobreescribe los cambios que el Hilo 20 realizó. A medida que se ejecutan los hilos, todos cargan, incrementan y guardan el valor de "contador" y sobrescriben los cambios que ellos han realizado.
Lo que necesitamos es una manera de evitar que los hilos se interrumpan mutuamente. La sentencia `contador = contador + 1`; debe ser autorizada para que se complete sin ser interrumpida. Puede hacer esto utilizando una característica denominada *exclusión mutua* o *mutex*. Nosotros podemos volver a la analogía de nuestro tren en este punto.

Utilizar la Exclusión Mutua para Gestionar el Uso de datos compartidos

Permitir que dos hilos compartan la misma variable, es un poco como, permitir que dos trenes compartan el mismo cantón ferroviario. Ambas acciones son peligrosas. El problema del cantón de la vía férrea, se resuelve utilizando un simple testigo de latón (en inglés, *token*) que posee el maquinista. Cuando el maquinista entra en el cantón, se le proporciona el testigo. Cuando el tren sale de la sección del carril único, el maquinista devuelve el testigo. Cualquier otro maquinista que quisiera entrar en el cantón, tiene que esperar su turno para recoger el testigo.

La exclusión mutua funciona de la misma manera usando una instancia de un objeto tomando el rol del testigo.

```
static object sincronizacion = new object();
```

Este objeto no contiene ningún dato. El objeto simplemente es utilizado como testigo y se encuentra “en manos” del proceso activo.

```
Monitor.Enter(sincronizacion);
contador = contador + 1;
Monitor.Exit(sincronizacion);
```

El código entre las llamadas a los métodos `Monitor.Enter` y `Monitor.Exit`, puede solamente ser realizado por un hilo al mismo tiempo. En otras palabras, no hay ninguna posibilidad de que Windows interrumpa la instrucción de incremento y cambie a otro proceso. Todas las operaciones de incremento se completarán en su totalidad. Una vez que la ejecución abandone las instrucciones existentes entre las llamadas a los métodos `Monitor`, el hilo puede ser detenido durante un tiempo como de costumbre.

Los hilos que están a la espera de ejecutarse, se encuentran “estacionados” en una cola de hilos en espera. Esta cola se forma y se mantiene por orden de llegada, por lo que el primer hilo en llegar al punto de entrada será el primero en obtener el testigo, mientras los otros hilos esperan su turno. La clase `Monitor` se encarga de todo esto, sin que nosotros tengamos que hacer nada, ni necesitemos saber cómo este proceso funciona.

Punto del Programador: Los Hilos pueden quebrantar completamente su programa

Si un hilo retiene un objeto de sincronización y no lo suelta, esto impedirá que otros hilos se ejecuten si necesitan ese objeto. Esta es una forma de que su programa falle. Otra de ellas, es el denominado "Bloqueo mutuo" (también conocido como interbloqueo, traba mortal, *deadlock*, *deadly embrace* o abrazo mortal) donde el hilo "`a`" posee el objeto "x" y está esperando el objeto "y", y el hilo "`b`" posee el objeto "y" y está esperando el objeto "x". Esta es la versión informática de "No voy a llamarle para disculparme, voy a esperar a que me llame él".

5.4.6 Control de Hilos

Hay una serie de características adicionales disponibles para gestionar hilos. Estas le permiten pausar los hilos, esperar a que finalice un hilo, ver si un hilo sigue activo y detenerlo y/o destruirlo.

Pausar Hilos

Como se pudo ver en el caso anterior, una forma de "pausar" un hilo, es creando un bucle con un valor límite enorme. Esto sin duda pausará su programa, a expensas de que otro hilo sea ejecutado. Si lo que quiere es pausar su programa, quizás para permitir al usuario leer la salida o esperar unos instantes a que algo suceda, debe utilizar el método `Sleep` proporcionado por la clase `Thread`:

```
Thread.Sleep(500);
```

Este método espera que se le suministre el número de milisegundos (milésimas de segundo), que se desea pausar la ejecución. El código anterior pausaría un programa por medio segundo.

Sincronizar Hilos

Es posible que quiera hacer que un hilo espere a que otro finalice su trabajo. Un hilo finaliza cuando sale del método desde el que fue invocado cuando se inicializó. En nuestros ejemplos anteriores, cada hilo terminará cuando finalice la llamada del método `bucleOcupado`. Una instancia de la clase `Thread` proporciona un método `Join,` al que se puede invocar para esperar a que otro hilo complete la operación.

```
t1.Join();
```

Esto haría que el hilo en ejecución quede a la espera de que el hilo `t1` finalice.

Control de Hilos

La clase `Thread` proporciona un conjunto de métodos que puede utilizar para controlar la ejecución de un hilo. Puede detener un hilo invocando al método `Abort`:

```
t1.Abort();
```

Esto le pide al sistema operativo que destruya ese hilo, y lo elimine de la memoria.

Si no quiere destruir un hilo, sino simplemente detenerlo durante un tiempo, puede invocar al método `Suspend`:

```
t1.Suspend();
```

El hilo se echará a dormir hasta que llame al método `Resume`.

```
t1.Resume();
```

Conocer el estado actual de un Hilo

Puede conocer el estado actual de un hilo utilizando la propiedad `ThreadState`. Esta enumeración tiene una serie de valores posibles.

Los de mayor utilidad son:

`ThreadState.Running`	el hilo se está ejecutando
`ThreadState.Stopped`	el hilo ha terminado de ejecutar el método
`ThreadState.Suspend`	el hilo ha sido detenido
`ThreadState.WaitSleepJoin`	El hilo está bloqueado, esperando a unirse a otro hilo, o esperando a un objeto Monitor

Como ejemplo, para mostrar si el hilo `t1` se está ejecutando:

```
if ( t1.ThreadState == ThreadState.Running )
{
   Console.WriteLine("Hilo ejecutándose");
}
```

5.4.7 Mantener la calma y no volverse loco con los Hilos

Los hilos son muy útiles. Un hilo permite que su programa se ocupe de una serie de tareas encadenadas mediante hilos de ejecución. La mejor forma de crear un sistema que soporte muchos usuarios, por ejemplo, un servidor web, es creando una aplicación multi-hilo que inicie un hilo por cada solicitud entrante. Esto hace que el código sea más fácil de manejar, y también que su programa pueda hacer un mejor uso de la potencia de procesamiento disponible.

Sin embargo, los hilos también pueden ser una gran fuente de frustraciones. Sabemos que nuestros programas a veces producen errores cuando se ejecutan. Esto ocurre principalmente porque hemos hecho algo mal. Cuando tenemos un error, analizamos lo que ocurre, y luego averiguamos cuál es la causa del problema. Hasta el momento, nuestros programas han sido ejecutados en un solo hilo y nosotros hemos podido someter a prueba a los datos que causan el problema.

Desafortunadamente, cuando se añaden multi-hilos a una solución esto ya no sucede. Un programa que funciona bien durante el 99,999% del tiempo puede fallar de manera fulminante, o simplemente bloquearse por completo, Cuando se produce un conjunto específico de eventos relacionados en el tiempo. Los problemas de sincronización como el "bloqueo mutuo" descrito anteriormente, solo pueden aparecer cuando dos usuarios solicitan una característica en particular al mismo tiempo.

Cuando se presentan problemas en un sistema multi-hilo lo más difícil siempre es hacer que el fallo ocurra para poder corregirlo. A menudo, la única forma de investigar el problema es añadir muchas instrucciones de escritura para que el programa cree un registro que pueda examinar después de que haya fallado. No obstante, el tiempo empleado en escribir el registro (log) de salida, afecta a la sincronización de los eventos en el programa, y a menudo puede provocar que un fallo se mueva o, si tiene mucha suerte, que desaparezca por completo.

Usted debe proteger las variables compartidas entre los hilos utilizando la clase `Monitor` como se ha descrito anteriormente. Esto evitará la corrupción inadvertida de datos. Si en ocasiones las variables presentan fallos de sincronización, tomando valores que aumentan y decrementan de forma ilógica, esta es una señal segura de que tiene a varios hilos peleándose por los mismos elementos de datos.

Debe tratar de evitar situaciones de “bloqueo mutuo”, haciendo que sus hilos sean *productores* o *consumidores* de datos. Si los consumidores están siempre esperando a los productores (y los productores nunca esperan a los consumidores), entonces puede estar seguro que nunca se producirá una situación en donde un hilo espere a otro que también lo esté esperando.

Finalmente, debe diseñar el comportamiento de registro y depuración (siempre puede utilizar la compilación condicional para desactivar el registro más tarde), de modo que si tiene un problema sea más sencillo obtener alguna información de diagnóstico, que le indique lo que su programa estaba haciendo en el momento en que falló.

Punto del Programador: Establezca los Hilos en su Diseño

Lo que realmente estoy diciendo aquí, es que cualquier utilización de hilos que vaya a hacer, debe ser valorada y establecida en la etapa de diseño de su programa y no posteriormente. Si quiere utilizar hilos, la manera en las que ellos van a gestionarse y cómo se comunican, deben ser decididas justo en la fase inicial de su diseño, y todo su sistema debe construirse teniéndolos a ellos siempre en cuenta.

5.4.8 Hilos y Procesos

Puede considerar a los diferentes hilos que conviven en un sistema, de la misma manera en la que un número de trenes comparten una vía férrea particular. El siguiente paso es tener más de una vía férrea. En términos de programación, este es un paso de hilos a procesos. Los procesos son diferentes a los hilos, ya que cada proceso tiene su propio espacio de memoria que está aislado de otros procesos.

Cuando usa un procesador de texto y un navegador web al mismo tiempo en su computadora, cada uno de ellos se ejecuta en su computadora como un proceso diferente. Su procesador de textos puede lanzar varios hilos que se ejecutan dentro de él (quizás uno para realizar la revisión ortográfica), pero nada en el procesador de textos puede acceder directamente a las variables del navegador.

En un programa de C#, puede crear un proceso e iniciarlo de forma similar a iniciar un hilo. También puede utilizar la clase `Process` para iniciar programas del sistema desde su propio código. Esta clase se encuentra disponible en el espacio de nombres `System.Diagnostics`, por lo que puede añadir una instrucción `Using` para facilitar el acceso a la clase:

```
using System.Diagnostics;
```

Para iniciar un programa en el sistema, puede utilizar el método `Start` de la clase `Process`:

```
Process.Start("Notepad.exe");
```

El método `Start` espera que se le suministre el nombre del programa que desea iniciar. La línea de código de C# anterior, iniciaría el programa Bloc de notas.

También puede crear instancias de `Process` que puede controlar desde su programa de manera similar a los hilos.

5.5 Manejo de errores estructurados

Por ahora, espero que esté reflexionando profundamente sobre la manera en que fallan los programas. Cuando construya un sistema también debería reflexionar sobre cómo va a gestionar la forma en que este fallará. Esta es en realidad una parte muy importante del proceso de diseño. Cuando algo inesperado ocurra, el programa debe lidiar con ello de manera eficaz. La clave para lograrlo es pensar en crear sus propias excepciones personalizadas para su sistema.

Anteriormente hemos visto cómo capturar errores y lanzar excepciones del sistema. Si algo inesperado sucede mientras se lee un archivo, o un método `Parse` recibe una cadena no válida, el sistema lanzará una excepción para indicar que no está conforme. Esto significa que un código potencialmente mal intencionado como este tiene que estar encerrado en una construcción `try` – `catch` para que nuestro programa pueda responder con sensatez. Ahora vamos a analizar más detalladamente las excepciones, y a aprender cómo crear nuestros propios tipos de excepciones.

5.5.1 La clase Exception

El espacio de nombres del sistema C# contiene una gran cantidad de excepciones diferentes, pero todas ellas están basadas en la clase padre `Exception`. Si quiere lanzar sus propias excepciones, se recomienda crear un tipo de excepción que extienda de la del sistema. Usted puede generar una excepción `System.Exception` cuando algo inesperado ocurra, pero esto significa que las excepciones producidas por su código se mezclarán con las producidas por otras partes del sistema. Si quiere que su código pueda manejar explícitamente sus errores, la mejor manera de hacerlo es crear sus propias excepciones personalizadas.

Esto significa, que, junto a todos los demás elementos y comportamientos de su sistema, tendrá que diseñar cómo su programa generará y manejará los errores. Todo esto (por supuesto, nos lleva de vuelta a los metadatos recopilados en la especificación, que le proporcionará la información que necesita acerca de cómo los comportamientos de los elementos pueden fallar y la forma en que los errores se producen.

5.5.2 Crear su propio tipo de excepción

Crear su propio tipo de excepción es muy sencillo. Esto se puede hacer simplemente extendiendo la clase `System.Exception`:

```
public class ExcepcionBancaria : System.Exception
{
}
```

Este código funciona porque la clase `System.Exception` cuenta con un método constructor predeterminado que no requiere de parámetros. El constructor predeterminado

`ExcepcionBancaria` (que es añadido automáticamente por el compilador), solamente puede utilizarse para crear una instancia de `Exception`.

Sin embargo, existe una versión del constructor `Exception`, que nos permite añadir un mensaje a la excepción. Esta versión puede tomarse y utilizarse para descubrir lo que salió mal. Para crear una excepción estándar con un mensaje, debo pasar al método constructor una cadena de texto. Si quiero utilizar esta versión del constructor en mi excepción bancaria, tengo que ordenar el encadenamiento del constructor para mi clase de excepción, y escribir el código de la siguiente manera:

```
public class ExcepcionBancaria : System.Exception
{
   public ExcepcionBancaria (string mensaje) : base (mensaje)
   {
   }
}
```

Esta version del método constructor, utiliza la palabra reservada `base` para invocar al método constructor de la clase padre y pasarle la cadena de texto del mensaje.

5.5.3 Lanzar una Excepción

La palabra reservada `throw` lanza una excepción en ese punto del código. El objeto de excepción que se lanza debe estar basado en la clase `System.Exception`. Por ejemplo, podría lanzar una excepción si el nombre del titular de una cuenta contuviera una cadena vacía. Podría hacerlo con este código:

```
if ( nombreEn.Length == 0 )
{
   throw new ExcepcionBancaria( "Nombre no válido" );
}
```

La palabra reservada `throw` viene seguida de una referencia a la excepción que se lanzará. En el código anterior, creo una nueva instancia de la excepción bancaria, y a continuación, la lanzo. En ese punto, la ejecución será transferida a la construcción `catch` “más cercana”. Esta excepción podría ser capturada por usted mismo, o por el propio sistema. Si la excepción es capturada por el propio sistema, mi programa finalizará. No obstante, usted puede capturar la excepción de la siguiente manera:

```
Cuenta a;

try
{
    a = new Cuenta(nuevoNombre, nuevaDireccion);
}
catch (ExcepcionBancaria excepcion)
{
    Console.WriteLine("Error: " + excepcion.Message);
}
```

El código intenta crear una nueva cuenta. Si al hacerlo se produce una excepción, el código que maneja la excepción es ejecutado. La referencia `exception` es establecida para hacer referencia a la excepción producida. El miembro `Message` de una excepción es el texto que fue proporcionado. Esto significa que, si yo intento crear una cuenta con nombre en blanco, la excepción será lanzada, la captura invocada, y el mensaje impreso por pantalla.

Punto del Programador: Diseñe sus propias excepciones de error

Una excepción es un objeto que describe las cosas que salieron mal. Este puede contener un mensaje de texto que puede mostrar al usuario para explicarle el problema que se ha producido. Sin embargo, debería considerar seriamente la posibilidad de ampliar la clase de excepción para crear excepciones de error que sean aún más informativas. Los errores deben ser numerados, es decir, las excepciones deben ser etiquetadas con un número de error. Esto sirve de gran ayuda a la hora de lidiar con diferentes idiomas. Si el cliente indica que ha recibido el "Error número 25" a la hora de reportar una incidencia, facilitará mucho el trabajo del personal de soporte a la hora de proporcionarle una solución. Tenga en cuenta que, al igual que ocurre con todo lo demás, también necesita diseñar su propio manejo de excepciones y errores.

5.5.4 Múltiples tipos de Excepciones diferentes

Vale la pena dedicar tiempo a pensar cómo se deben gestionar las excepciones y los errores en su sistema. Nosotros podríamos incluir muchas construcciones de captura, si así lo queremos, teniendo en cuenta que cada una de esas construcciones de captura estará correlacionada con el tipo de excepción lanzada:

```
public class ExcepcionBancariaNombreNoValido : System.Exception
{
   public ExcepcionBancariaNombreNoValido (string mensaje) : base (mensaje)
   {
   }
}

public class ExcepcionBancariaDireccionNoValida : System.Exception
{
   public ExcepcionBancariaDireccionNoValida (string mensaje) :
      base(mensaje)
   {
   }
}
```

Ahora puedo utilizar diferentes construcciones de captura, para mostrar un mensaje personalizado dependiendo del error que se haya producido en el programa:

```
Cuenta a;

try
{
    a = new Cuenta("Rob", "");
}
catch (ExcepcionBancariaNombreNoValido nombreExcepcion)
{
    Console.WriteLine("Nombre no válido: " + nombreExcepcion.Message);
}
catch (ExcepcionBancariaDireccionNoValida direcExcepcion)
{
    Console.WriteLine("Dirección no válida: " + direcExcepcion.Message);
}
catch (System.Exception excepcion)
{
   Console.WriteLine("Excepción del sistema: " + excepcion.Message);
}
```

Cada una de las capturas coincide con un tipo diferente de excepción. Al final de listado de las construcciones de captura, incluyo un bloque destinado a capturar la excepción del sistema. Este será invocado, si la excepción no es un nombre o una dirección.

Punto del Programador: Los programas fallan frecuentemente en los controladores de errores

Si lo piensa con detenimiento, los controladores de errores son bastante difíciles de testear. Estos solamente se ejecutan cuando sucede algo inesperado, y en la mayoría de ocasiones solamente se verán sometidos a pruebas que asumen de que todo funciona correctamente. Esto significa que es más probable que los errores se presenten en los controladores de errores, ya que el código no se emplea con tanta asiduidad como el del resto del sistema. Por supuesto, esta es una solución que tiene como objetivo evitar grandes desastres, ya que el controlador/manejador de errores se supone que está ahí para resolver este tipo de incidencias, y si esta falla, generalmente la situación se volverá mucho más fea.

Como programador profesional, debe asegurarse de testear su código de control de errores de manera tan exhaustiva como el del resto de su sistema. Esto podría significar el que tenga que crear versiones especiales de prueba del sistema para forzar a provocar errores dentro del código. Créame, ¡valdrá la pena el esfuerzo!

Las características del controlador de errores deben ser diseñadas. Cuando crea el sistema, debe considerar cuántos errores y de qué tipos, va a tener que gestionar su sistema.

5.6 Organización del código fuente del programa

Hasta el momento, nosotros hemos establecido por completo el código fuente de nuestro programa en un solo archivo, que posteriormente compilamos y ejecutamos. Esto está bien para pequeños proyectos, pero ahora estamos empezando a escribir programas mucho más extensos y resulta cada vez más necesario organizar los elementos de una forma más adecuada. Esto es especialmente importante cuando el trabajo de un proyecto es llevado a cabo por un equipo de varias personas.

Para lograr esto, nosotros tenemos que resolver dos problemas:

- cómo vamos distribuir *físicamente* el código en una serie de archivos.
- cómo vamos a identificar *lógicamente* los elementos en nuestro programa.

Los dos problemas son distintos e independientes, y C# proporciona mecanismos para resolver ambos. En esta sección vamos a ver cómo un gran programa se puede dividir en varios fragmentos diferentes.

5.6.1 Utilizar Archivos de Código Independientes

En un sistema de tamaño considerable, un programador querrá desplegar y difundir a través de varios archivos de código fuente diferentes. Cuando diseñe su solución a un problema, debe decidir dónde ubicar todos los archivos.

En nuestro programa de administración bancaria, hemos identificado la necesidad de tener una clase para realizar un seguimiento de una cuenta en particular. La clase será llamada `Cuenta`. Considere una clase `Cuenta` realmente simple:

```
public class Cuenta
{
    private decimal saldo = 0;

    public void IngresarEfectivo( decimal cantidad )
    {
       saldo = saldo + cantidad;
    }

    public decimal ObtenerSaldo()
    {
       return saldo;
    }

    public bool RetirarEfectivo( decimal cantidad )
    {
       if ( cantidad < 0 )
       {
          return false;
       }

       if (saldo >= cantidad )
       {
          saldo = saldo - cantidad;
          return true;
       }
       else
       {
          return false;
       }
    }
}
```

Esta es una clase con un comportamiento adecuado, ya que no nos permite retirar más dinero del que tenemos en la cuenta. Podríamos incluirla dentro de un archivo llamado "`Cuenta.cs`", si así lo quisiéramos. Sin embargo, es posible que deseemos crear muchas otras clases que se ocupen de lidiar con las cuentas bancarias. Así que en vez de eso he decidido incluir la clase en un archivo

llamado "`GestionDeCuentas.cs`". Cuando yo añada más elementos de cuentas para gestionar, podré incluirlas dentro de este archivo. El problema es que ahora el compilador nos indica un error cuando compilamos el archivo:

```
error CS5001: GestionDeCuentas.exe' no tiene un punto de entrada definido
```

El compilador espera producir un programa ejecutable. Éstos se caracterizan porque tienen un punto de entrada definido en forma de método `Main`. Nuestra clase de cuenta bancaria no tiene un método principal, debido a que el programa nunca se iniciará en realidad ejecutando una cuenta. Por lo tanto, el compilador no puede crear un archivo ejecutable, ya que no sabe dónde comienza el programa. No obstante, otros programas crearán y ejecutarán cuentas cuando las necesiten utilizando este archivo de código independiente.

Tenga en cuenta que he declarado la clase `Cuenta`, utilizando el modificador de acceso `public`. Esto se hace así, para que clases existentes en otros archivos puedan hacer uso de ella. Puede aplicar los niveles de protección a las clases, de la misma manera, en que puede proteger a los miembros de la clase. Por regla general, si quiere que sus clases sean utilizadas en bibliotecas, deben ser declaradas utilizando el modificador de acceso `public`.

Crear una Biblioteca de Clases

Puedo resolver el anterior problema de compilación, pidiéndole al compilador que produzca una *biblioteca*, en vez de un ejecutable. Para ello, utilizo la opción `target` proporcionada por el compilador. Las opciones del compilador, nos permiten modificar lo que hace un comando. El compilador sabe que le estamos especificando una opción, porque éstas siempre comienzan con un carácter de barra diagonal `/`:

```
csc /target:library GestionDeCuentas.cs
```

El compilador no buscará ahora un método `Main`, porque le hemos indicado que produzca una biblioteca en lugar de un programa. Si reviso lo que ha sido creado, comprobaré que tal y como esperaba, el compilador no ha creado un archivo ejecutable sino un archivo de biblioteca:

```
GestionDeCuentas.dll
```

La extensión `dll` significa *biblioteca de enlace dinámico*. Lo que significa que el contenido de este archivo, se cargará *dinámicamente* a medida que se ejecuta el programa.

Utilizar una Biblioteca de Clases

Ahora que ya tengo una biblioteca, lo siguiente que tengo que hacer es averiguar cómo utilizarla. Primeramente, voy a crear otro archivo de código fuente llamado `ProbarCuenta.cs`. Este contiene un método `Main`, que utilizará la clase `Cuenta`:

```
using System;

class ProbarCuenta
{
    public bool static void Main()
    {
       Cuenta pueba = new Cuenta();
       prueba.IngresarEfectivo(50);
       Console.WriteLine("Saldo:" + prueba.ObtenerSaldo());
    }
}
```

Este código crea una nueva cuenta, ingresa 50 libras dentro de ella, y a continuación, imprime el saldo disponible en la cuenta por pantalla. Si probara a compilar este archivo, recibiré una gran cantidad de mensajes de error por parte del compilador:

```
ProbarCuenta.cs(5,3): error CS0246: No se puede encontrar el tipo o el
nombre de espacio de nombres 'Cuenta' (¿falta una directiva using o una
referencia de ensamblado?)

ProbarCuenta.cs(6,3): error CS0246: No se puede encontrar el tipo o el
nombre de espacio de nombres 'prueba' (¿falta una directiva using o una
referencia de ensamblado?)

ProbarCuenta.cs(7,37): error CS0246: No se puede encontrar el tipo o el
nombre de espacio de nombres 'prueba' (¿falta una directiva using o una
referencia de ensamblado?)
```

El problema es que el compilador no sabe dónde ir a buscar el archivo `GestionDeCuentas.cs`, para encontrar y utilizar la clase `Cuenta`. Esto tiene como consecuencia, el que no pueda crear la prueba, lo que causa todavía más errores.

Para resolver el problema, necesito remitir al compilador al archivo `GestionDeCuentas.dll` para que de esta forma pueda encontrar y utilizar la clase `Cuenta`:

```
csc /reference: GestionDeCuentas.dll ProbarCuenta.cs
```

A la opción de la referencia, le sigue la lista de archivos de la biblioteca que se utilizarán. En este caso, solamente hay que indicar la referencia al archivo de biblioteca que contiene la clase requerida. El compilador ahora sabe dónde encontrar todas las partes de la aplicación, y, por tanto, puede crear el programa ejecutable.

Referencias de biblioteca en tiempo de ejecución

Ahora tenemos dos archivos que contienen el código del programa:

`GestionDeCuentas.dll`	la biblioteca que contiene el código fuente de la clase `Cuenta`
`ProbarCuenta.exe`	el programa ejecutable que crea una instancia de `Cuenta`

Ambos archivos deben estar presentes para que el programa funcione correctamente. Esto se debe a la "dinámica" de la *biblioteca de vínculos dinámicos*. Lo que significa que la biblioteca solo se carga cuando se ejecuta el programa, no cuando se genera.

Eliminar Archivos de Componentes de nuestro Sistema

Esto significa que si yo hago alguna acción horrenda como por ejemplo eliminar el archivo `GestionDeCuentas.dll`, y a continuación, ejecuto el programa, provoco que ocurran toda clase de cosas nefastas:

```
Excepción no controlada: System.IO.FileNotFoundException: Archivo o nombre
de ensamblado GestionaDeCuentas, o una de sus dependencias, no fue
encontrada.
Nombre de archivo: "GestionDeCuenta" en ProbarCuenta.Main()
… y un montón de otras cosas que nos afectan
```

Debido a esto, debemos tener cuidado cuando a la hora de publicar un programa, y siempre asegurarnos de que todos los archivos de componentes estén presentes cuando se ejecuta el programa.

Actualizar Archivos de Componentes de nuestro Sistema

La creación de un sistema a partir de una serie de componentes ejecutables, tiene la ventaja de que podemos actualizar una parte sin afectar a todo lo demás. Si yo modifico la clase `GestionDeCuentas`, y vuelvo a compilarla, la nueva versión es detectada y utilizada por la clase `ProbarCuenta` automáticamente. Por supuesto, esto sólo funciona siempre y cuando no cambie la apariencia de las clases o métodos que la clase `ProbarCuenta` utiliza.

La buena noticia, es que puedo corregir las partes corruptas del programa sin tener que publicar una nueva versión completa.

La mala noticia es que esto suele provocar problemas. Hay un término especial, "infierno de las dll" (dll hell), referido a las complicaciones que surgen al trabajar con bibliotecas de vínculos dinámicos. A menos que el código corregido corresponda **precisamente y únicamente** a este punto del programa, existe una buena posibilidad de que pueda corromper alguna otra parte del programa que lo utilice. Windows trabaja precisamente con bibliotecas de vínculos dinámicos, y debido a esto, el número de veces que al instalar un nuevo programa (o peor aún, una actualización de uno ya instalado con anterioridad), ha provocado que otro programa de mi computadora dejara de funcionar, es demasiado grande como para recordarlo.

Punto del Programador: Utilice el Control de Versiones y la Gestión de cambios

Muchas cosas se encuentran vinculadas a la necesidad de contar con una buena planificación y gestión de cambios. Y aquí estamos una vez más. Cuando piense en vender su aplicación para ganar dinero, debe asegurarse de contar con enfoque de gestión centralizado en cómo va a enviar las actualizaciones y correcciones de su sistema, a los clientes que haya adquirido su programa. La buena noticia es que hay formas de asegurarse de que ciertas versiones de su programa sólo funcionen con determinados archivos particulares. La mala noticia es que usted tiene que planificar cómo utilizar esta tecnología, y luego asegurarse de utilizarla correctamente. Si esto le suena aburrido y pedante entonces lo siento mucho, pero siento comunicarle que, si no sigue esta recomendación, se volverá loco o entrará en bancarrota. O lo que es todavía peor, ambas cosas.

5.6.2 Espacio de nombres

Nosotros podemos utilizar archivos de biblioteca para dividir nuestra solución en varios archivos. Esto hace que realizar la gestión de la solución sea un poco más sencillo. Pero nosotros también tenemos otro problema al respecto. No queremos solamente dividir los elementos en fragmentos de información físicos (denominados, *chunk*), sino que además queremos dividirlos en partes *lógicas* también.

Si no está seguro de lo que quiero decirle con esto, considere la situación en nuestro banco. Nosotros hemos decidido que `Cuenta` es un nombre sensato para la clase que almacena toda la información de una cuenta cliente.
Pero si considera todas las operaciones bancarias que nuestro banco puede llegar a realizar, descubrirá que la palabra "cuenta", se presenta de manera imprevista en otras partes del programa que no hemos considerado. El banco comprará cosas como clips, sellos de caucho, bolígrafos con peana (pre-suministrados sin tinta por supuesto) y similares, a proveedores.

Podría decirse que el banco tiene una cuenta con dichos proveedores. Es muy posible que el director gerente del banco desee realizar un seguimiento de estas cuentas mediante un sistema informático. Tal vez los programadores podrían decidir que un nombre sensato para tal cosa sería el de `Cuenta`. ¡Arrgh!, ahora nos dirigimos al verdadero problema. Si los dos sistemas se juntan, nosotros podemos esperar una batalla digital hasta la muerte a cuenta de lo que realmente significa "Cuenta". La cual sería catastrófica para nuestros intereses.

Podríamos resolver el problema, cambiando el nombre de nuestra clase `Cuenta` de clientes. Pero esta solución terminaría siendo un poco caótica, e implicaría que en tiempo de diseño tendríamos que asegurarnos de nombrar todas nuestras clases de una manera que siempre fuera única.

Una solución mucho más adecuada, sería indicar que nosotros disponemos de un espacio de nombres `ClienteBancario` en la que la palabra `Cuenta` tiene un significado particular. Nosotros también podemos tener un espacio de nombres `ProveedorDeSuministros`. Esto evita que los dos nombres entren en conflicto, ya que no están definidos en el mismo espacio de nombres.

Establecer una Clase dentro de un Espacio de Nombres

Hasta ahora, todos los nombres que hemos utilizado han sido creados en lo que se denomina el espacio de nombres *global*. Esto se debe a que no hemos establecido explícitamente un espacio de nombres en nuestros archivos de código fuente. Sin embargo, son muy fáciles de establecer:

```
namespace ClienteBancario
{
   public class Cuenta
   {
       private decimal saldo = 0;

       public void IngresarEfectivo( decimal cantidad )
       {
          Saldo = saldo + cantidad;
       }

       public decimal ObtenerSaldo()
       {
          return saldo;
       }

       public bool RetirarEfectivo( decimal cantidad )
       {
          if ( cantidad < 0 )
          {
             return false;
          }

          if (saldo >= cantidad )
          {
             saldo = saldo - cantidad;
             return true;
          }
          else
          {
             return false;
          }
       }
   }
```

```
}
```

He utilizado la palabra reservada `namespace` para establecer el espacio de nombres. A esto le sigue un bloque de clases. Todas las clases declaradas en este bloque se consideran como parte del espacio de nombres especificado, en este caso `ClienteBancario`.
Un archivo de código fuente puede contener muchos espacios de nombres, y cada uno de ellos puede contener tantas clases como desee.

Utilizar una Clase desde un Espacio de Nombres

Una clase Global (es decir, una creada fuera de cualquier espacio de nombres), solamente puede ser referida a través de su nombre. Si quiere utilizar una clase desde un espacio de nombres, también debe especificar el espacio de nombres donde se encuentra.

```
ClienteBancario.Cuenta prueba;
```

Este código crea una variable que puede hacer referencia a instancias de la clase `Cuenta`. La clase `Cuenta` que nosotros utilizamos es la que se encuentra en el espacio de nombres `ClienteBancario`.

Un nombre como este, que presenta el espacio de nombres delante, es conocido como un nombre *completamente cualificado*. Si nosotros queremos crear una instancia de clase, utilizaremos de nuevo su nombre completo cualificado:

```
prueba = new ClienteBancario.Cuenta();
```

Si yo quiero utilizar la clase `Cuenta` del espacio de nombres `ClienteBancario`, tengo que modificar mi clase de prueba en consecuencia.

Utilizar un Espacio de nombres

Si está utilizando muchos elementos de un espacio de nombres determinado, C# proporciona una manera en la que puede indicarle al compilador que busque en ese espacio de nombres siempre que tenga que resolver el nombre de un elemento determinado. Nosotros hacemos esto con la palabra reservada `using`:

```
using ClienteBancario;
```

Cuando el compilador llega a una línea como:

```
Cuenta CuentaDeRob;
```

- automáticamente busca en el espacio de nombres `ClienteBancario` para ver si dentro de éste existe una clase llamada `Cuenta`. Si la hay, la utiliza. Nosotros ya hemos utilizado muchas veces

esta técnica. El espacio de nombres `System`, es donde se encuentran muchos de los componentes de la biblioteca. Nosotros podemos utilizarlo mediante su nombre completo cualificado:

```
System.Console.WriteLine( "Hola Mundo" );
```

Sin embargo, lo más común es que los programas lo utilicen conteniendo la línea

```
using System;
```

- en la parte superior del código. Esto significa que el programador solamente tiene que escribir:

```
Console.WriteLine( "Hola Mundo" );
```

El compilador se dice así mismo, "ah, iré a buscar un elemento llamado `Console` en todos los espacios de nombres que me han dicho que utilice". Si encuentra uno, y solamente un elemento `Console`, entonces todo va bien y lo utiliza. Si encuentra dos elementos `Console` (Si fuera un idiota, podría incluir una clase `Console` en mi espacio de nombre `ClienteBancario`), nos indicará que no sabe cuál de ellos utilizar.

Anidar Espacios de nombres

Usted puede incluir un espacio de nombres dentro de otro. Esto le permite dividir los elementos en fragmentos de información físicos más pequeños (*chunks*). El espacio de nombres `System` es así, e incluye un espacio de nombres que forma parte del espacio de nombres del sistema que está específicamente relacionado con la Entrada/Salida. Puede utilizar los elementos que se encuentran dentro del espacio de nombres `System.IO`, incluyendo la instrucción `using` delante del nombre del espacio, en la parte superior del código fuente:

```
using System.IO;
```

En términos de declarar el espacio de nombres, puede hacerlo de la siguiente manera:

```
namespace ClienteBancario
{
   namespace Cuentas
   {
       // las clases de cuenta van aquí
   }
   namespace EstadoDeCuenta
   {
       // las clases del estado de cuenta van aquí
   }
   namespace CartasDesagradables
   {
```

```
        // las clases de cartas desagradables van aquí
    }
}
```

Ahora, puedo utilizar las clases según lo requiera:

```
using ClienteBancario.CartasDesagradables;
```

Por supuesto, los espacios de nombres que utilice siempre deben ser diseñados cuidadosamente. Pero probablemente, ya se había dado cuenta de esto.

5.6.3 Espacios de Nombres en Archivos independientes

No hay ninguna regla que indique que deba establecer todas las clases de un espacio de nombres particular en un único archivo determinado. Es perfectamente válido distribuir las clases en varios archivos fuente diferentes. Debe asegurarse de que los archivos que desea utilizar contengan todas las partes necesarias, y esto significa, por supuesto, un mayor grado de planificación y organización.

Punto del Programador: Los Nombres Completamente Cualificados son más adecuados de utilizar

Hay dos maneras de poder acceder a un elemento. Puede hacerlo especificando la ubicación completa del elemento utilizando un Nombre Completamente Cualificado (`ClienteBancario.CartasDesagradables.AvisoCuentaAlDescubierto`), o especificando simplemente el nombre del espacio mediante el uso de la instrucción `using`. De las dos maneras disponibles de hacer esto, mi opción preferida es la del nombre completamente cualificado. Sé que esto hace que los programas sean un poco más complejos de escribir, pero por otra parte cuando estoy leyendo el código puedo ver exactamente de dónde proviene un recurso especificado. Si solamente veo el nombre `AvisoCuentaAlDescubierto` en el código, yo no tengo ni idea de dónde proviene, por lo que, en caso de necesitarlo, tendré que buscar a través de todos los espacios de nombre que estoy utilizando para encontrarlo.

5.7 Interfaz Gráfica de Usuario

Los programas que hemos escrito hasta ahora usan una línea de comandos para interactuar con el usuario. Es decir, la computadora hace preguntas, el usuario ingresa sus respuestas, y la computadora finalmente las procesa para hacer lo que se le ha pedido. Ahora vamos a pasar al mundo de las Interfaces Gráficas de Usuario.

Una interfaz de usuario es un nombre pijo que se le da a lo que la gente realmente ve cuando utilizan su programa. Esta se compone de campos de texto, etiquetas e imágenes con las que el usuario interactúa para hacer su trabajo. Parte del trabajo del programador es crear este "front-end", y luego colocar los comportamientos apropiados "detrás de la pantalla", para permitir al usuario manejar el programa y que pueda realizar lo que necesita. En esta sección, vamos a descubrir cómo crear un programa que use una interfaz gráfica de usuario.

Vamos a crear una aplicación mediante un marco de interfaz de usuario llamado Windows Presentation Foundation (WPF) que utiliza un lenguaje de marcado llamado XAML para definir las páginas que ve el usuario. Este se encuentra disponible para sistemas operativos Windows, incluyendo Windows Vista, Windows 7 y el entorno de escritorio de Windows 8. Si desea crear aplicaciones para Windows RT, los fundamentos son similares, pero tendrá que realizar algunas modificaciones en los archivos XAML que cree.

Hay una parte en el Mago de Oz donde Dorothy, mirando el largo camino de flores existente en la mágica Tierra de Oz, se dirige a su perro y le dice “Totó, me parece que esto no es Kansas, ¿sabes?”. Nosotros en este momento, nos encontramos en una situación parecida dentro de C#. Esta sección no va a estar únicamente destinada al lenguaje de programación C#, ya que también vamos a utilizar XAML como lenguaje de marcado para crear interfaces de usuario. De todas formas, también vamos a utilizar el lenguaje C# para proporcionar los comportamientos requeridos. Podemos crear programas que utilicen una interfaz de usuario basada en XAML mediante otros lenguajes, como por ejemplo Visual Basic .NET.

Nosotros vamos a descubrir que es difícil hablar de XAML sin mencionar Visual Studio, el entorno de desarrollo integrado que permite a los programadores crear aplicaciones que contengan una interfaz de usuario diseñada en XAML y conectada mediante código a C#.

Visual Studio puede sincronizar la descripción XAML de una interfaz de usuario, con el código de programa que esté conectado a ella. También contiene un editor gráfico que se puede utilizar para diseñar la interfaz en sí.

Tal vez sea mejor considerar C#, XAML y Visual Studio como un todo, ya que fueron diseñados para trabajar conjuntamente facilitando a los programadores la creación de este tipo de aplicaciones.

Punto del Programador: Aprenda a utilizar Visual Studio

Es posible crear archivos XAML y aplicaciones completas, solamente escribiendo los archivos de texto que contienen los elementos de C# y XAML desde el propio bloc de notas. Sin embargo, esto tomaría mucho tiempo. En este punto, debe invertir algo de tiempo en aprender a utilizar Visual Studio, que proporciona un entorno muy rico para la creación de programas como este. Visual Studio le permitirá arrastrar elementos de visualización sobre una superficie de diseño (si debe colocar los elementos manualmente), y crear automáticamente controladores de eventos para los distintos elementos disponibles, como por ejemplo botones.

5.7.1 El Lenguaje de Marcado XAML

Nosotros vamos a utilizar el lenguaje eXtensible Application Markup Language (o XAML), traducido como Lenguaje de Marcado de Aplicaciones Extensible, diseñado por Microsoft para facilitar la creación de una aplicación atractiva. Para entender esto, vamos a tener que aprender algunas cosas acerca de cómo funcionan los lenguajes de marcado. La buena noticia es que este conocimiento es extremadamente útil, muchos sistemas modernos de interfaz de usuario funcionan de manera similar.

XAML es un lenguaje *declarativo*. Esto significa que todo lo que puede hacer es proporcionar información a una computadora sobre cosas. En el caso de XAML esta cosa es el diseño de la pantalla, o de la página, que el usuario visualiza.

Lenguajes de Marcado Extensibles

En este punto, es posible que se pregunte qué es realmente un Lenguaje de Marcado de Aplicaciones Extensible. Bueno, pues es un lenguaje de marcado para aplicaciones que es extensible. Seguro que esta explicación le ayudó. Lo que queremos decir con esto es que puedes usar las reglas del lenguaje para crear construcciones que describan cualquier cosa. El idioma español es muy parecido a esto. Nosotros tenemos letras y signos de puntuación que son los símbolos que utilizamos para escribir en español. También tenemos reglas (denominadas gramaticales) que establecen cómo componer palabras y oraciones, y tenemos diferentes tipos de palabras. Tenemos adjetivos que describen cosas y verbos que describen acciones. Cuando aparece alguna cosa nueva inventamos un nuevo conjunto de palabras para describirla. Alguien tuvo que inventar la palabra "computadora" cuando se inventó la computadora, junto con frases como "arranque", "bloqueo del sistema" y "demasiado lento".

Los lenguajes basados en XML son extensibles en el sentido de que podemos inventar nuevas palabras y frases que se ajusten a las reglas del lenguaje y utilizar estas nuevas construcciones para describir cualquier cosa que queramos. Estos son denominados lenguajes de *marcado* porque a

menudo se utilizan para describir la disposición de los elementos en una página. El marcado de palabras se utilizó originariamente en la impresión, cuando se quería decir cosas como "Imprime el nombre Rob Miles en una fuente muy grande". El lenguaje de marcado más famoso es probablemente HTML, HyperText Markup Language, que es utilizado por la World Wide Web para describir el formato de las páginas web.

Los programadores inventan con frecuencia sus propios formatos de almacenamiento de datos utilizando XML. Como ejemplo, un fragmento de código XML que describe un conjunto de máximas puntuaciones podría ser así:

```
<?xml version="1.0" encoding="us-ascii" ?>
<RegistroPuntuacionesMaximas count="2">
    <PuntuacionMaxima juego="Breakout">
        <nombrejugador>Rob Miles</nombrejugador>
        <puntuacion>1500</puntuacion>
    </PuntuacionMaxima>
    <PuntuacionMaxima juego="Space Invaders">
        <nombrejugador>Rob Miles</nombrejugador>
        <puntuacion>4500</puntuacion>
    </PuntuacionMaxima>
</RegistroPuntuacionesMaximas>
```

Este es el código de un pequeño archivo XML que describe un registro de puntuaciones máximas (records) para un sistema de videojuegos. El elemento `RegistroPuntuacionesMaximas` contiene dos elementos `PuntuacionMaxima`, uno para el videojuego `Breakout` y otro para el videojuego `Space Invaders`. Los dos elementos de puntuación máxima están contenidos dentro del elemento `RegistroPuntuacionesMaximas`. Cada uno de los elementos tiene una propiedad que da el nombre del juego y que a su vez contiene otros dos elementos más, el nombre del jugador y la puntuación que consiguió. Esto nos resulta muy fácil de entender. Visualizando el texto de arriba, no es difícil conocer la máxima puntuación que se logró alcanzar en el videojuego `Space Invaders`.

La línea en la parte superior del archivo, indica a quien quiera leer el archivo, la versión del estándar XML en la que está basado y la codificación de los caracteres utilizada en el archivo. XAML toma las reglas de un lenguaje de marcado extensible y las usa para crear un lenguaje que describa los componentes del contenido de una pantalla.

```
<TextBox Height="72" HorizontalAlignment="Left" Text="0"
VerticalAlignment="Top" Width="460" TextAlignment="Center" />
```

Si nosotros nos detenemos a leer la descripción anterior del elemento TextBox, podemos observar que los diseñadores de XAML han creado nombres de campos que coinciden con nuestros requerimientos. Nosotros vamos a conocer XAML con mayor detalle, en esta sección.

Esquema XML

El estándar XML también contiene descripciones sobre cómo crear un *esquema* que defina un formato de documento particular. Por ejemplo, en el esquema visto anteriormente para registrar la información de las puntuaciones máximas podría decirse que un elemento `PuntuacionMaxima` debe contener una propiedad de `nombrejugador` y otra de `puntuacion`. También podría decirse que un elemento como `PuntuacionMaxima` puede contener un valor de `Fecha` (la fecha en la que se consiguió alcanzar la puntuación más alta), pero que no sería requerido por cada valor de `PuntuacionMaxima`.

Este sistema estándar de esquema y formato, permite de manera sencilla a los desarrolladores crear formatos de datos para propósitos particulares. Este hecho es respaldado por la gran cantidad de herramientas de diseño que existen para crear documentos y esquemas. .NET framework incluso proporciona una forma mediante la cual un programa puede guardar un objeto como un documento XML formateado. De hecho, el archivo de solución de Visual Studio se almacena realmente como un documento XML.

Por lo que a nosotros respecta, vale la pena recordar que XML es útil para este tipo de cosas, pero por el momento quiero centrarme en el lenguaje XAML.

XAML y el diseño de página

Un archivo de XAML puede describir una página completa. Al crear una nueva aplicación WPF, obtendrá una página que solamente contiene algunos elementos. A medida que añade más elementos de descripción el archivo crece. Algunos elementos funcionan como *contenedores*. Lo que significa que pueden contener otros componentes. Estos son de gran utilidad cuando posicionar o acomodar elementos, por ejemplo, hay un elemento Rejilla que puede contener un conjunto de otros elementos en una disposición de cuadrícula. El archivo XAML también puede contener las descripciones de animaciones y transiciones que se pueden aplicar a los elementos de la página para crear interfaces de usuario aún más impresionantes. No vamos a dedicar demasiado tiempo a los aspectos de diseño de XAML; basta decir que puede crear interfaces de usuario impresionantes para sus programas con esta herramienta. También existe una herramienta de diseño profesional destinada especialmente para esta tarea llamada "Expression Blend”.

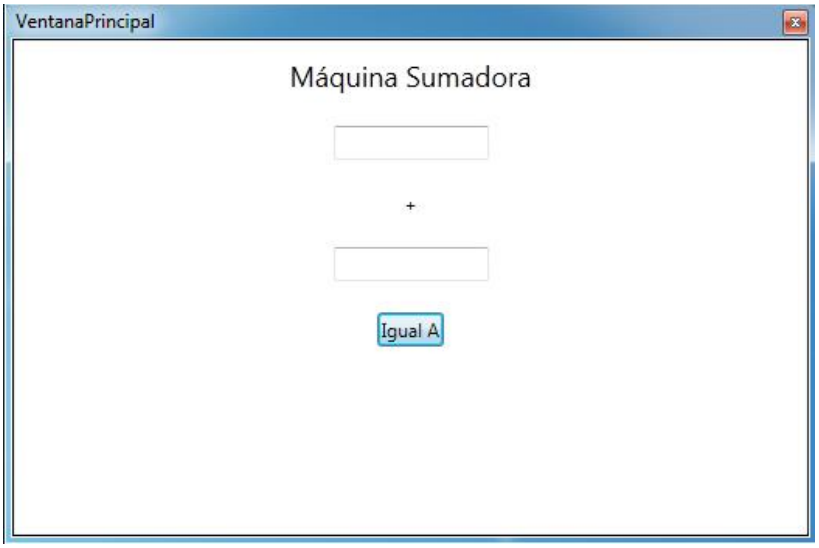
Desafortunadamente, resulta que la mayoría de los programadores (incluyéndome a mí), no son tan buenos diseñando interfaces de usuario atractivas (aunque estoy seguro de que este no es su caso). En la vida real, una empresa empleará diseñadores gráficos que crearán interfaces front ends desde un aspecto artístico. La función del programador será colocar el código detrás de estas pantallas para realizar el trabajo requerido.

Microsoft diseñó XAML como un reconocimiento de este problema. Este impone una separación muy fuerte entre el diseño de la pantalla y el código que la controla. Esto hace que sea fácil para

un programador crear una interfaz de usuario inicial, que posteriormente puede ser modifica por un diseñador que la haga mucho más atractiva. También es posible encontrarse con casos en los que un programador recibe un diseño completo de una interfaz de usuario, y tenga que establecer los comportamientos requeridos detrás de cada uno de los componentes de la pantalla.

Describir elementos XAML

Podemos comenzar a descubrir cómo el lenguaje XAML nos permite diseñar una aplicación, utilizándolo para construir un programa. Considere la siguiente pantalla.



Este es un programa de interfaz de ventana muy simple, a la que le he dado el nombre de “Máquina Sumadora”. Se puede utilizar para realizar sumas muy sencillas. Sólo tiene que introducir dos números en las dos cajas de texto de la parte superior y, a continuación, pulsar el botón “Igual a”, para obtener el resultado de la suma. Por el momento, no está mostrando más que, 0 más 0 es igual a 0. Cada elemento individual de la pantalla se denomina *UIElement* o Elemento de Interfaz de Usuario. Yo voy a denominarlos como elementos, a partir de este instante. Hay seis de ellos en la imagen de ejemplo de la “Máquina Sumadora”:

1. El título “Máquina Sumadora”. Este es un bloque de texto con un tamaño de fuente 18 para que destaque.

2. La caja de texto superior, donde puedo introducir un número.

3. Un elemento de texto que contiene el carácter +.

4. El elemento de texto inferior, donde puedo introducir otro número.

5. Un botón, al que puedo hacer clic para que realice la suma.

6. Una caja de texto de resultado, que cambia para mostrar el resultado cuando se presiona el botón. Por el momento está vacío, ya que aún no hemos hecho ninguna suma.

Cada uno de estos elementos tiene una posición y un tamaño de texto particular en la pantalla, y también muchas otras propiedades. Nosotros podemos cambiar el color del texto de una caja de texto, indicar que esté alineado a la izquierda, a la derecha o en el centro de la caja, y muchas otras cosas, actualizando el XAML que describe la página. El código XAML real, que especifica el diseño de la página es el siguiente:

```
<StackPanel>
    <TextBlock Text="Máquina Sumadora" TextAlignment="Center"
      Margin="0,10" FontSize="18"></TextBlock>
    <TextBox Name="primerNumeroTextBox" Width="100"
      Margin="0,10" TextAlignment="Center"></TextBox>
    <TextBlock Text="+" TextAlignment="Center" Margin="0,10"></TextBlock>
    <TextBox Name="segundoNumeroTextBox" Width="100"
      Margin="0,10" TextAlignment="Center"></TextBox>
    <Button Content="Igual a" Name="botonIgualA"
      HorizontalAlignment="Center" Margin="0,10"
      Click= ="botonIgualA_Click"></Button>
    <TextBlock Text="resultadoTextBlock" Text=""
      TextAlignment="Center" Margin="0,10"></TextBlock>
</StackPanel>
```

Si revisa con detalle este código, seguro puede asignar cada uno de los elementos de la ventana a los elementos especificados en este archivo XAML. Lo único que puede confundirle es el elemento `StackPanel`. Este es muy simple, pero enormemente útil. En lugar de tener que definir la posición en la pantalla de cada uno de los elementos, un `StackPanel` nos permite "apilar" una serie de elementos de visualización de manera sencilla. La disposición predeterminada utilizada por el flujo de contenido es apilar los elementos verticalmente, pero también pueden apilarse en horizontal. También podemos, y esto es realmente útil, anidar paneles `StackPanel`, es decir, utilizar un panel de diseño como elemento secundario, para que contenga elementos en paralelo formando una pila.

Punto del Programador: Utilice la disposición automática tanto como pueda

Siempre estoy preocupado cuando empiezo a colocar los elementos en la pantalla. Tan pronto como usted tenga que hacerlo, comenzará a hacerse suposiciones sobre las dimensiones de la pantalla que está utilizando y el tamaño del texto. Las computadoras modernas se suministran en una amplia gama de diferentes dimensiones de pantalla, y los usuarios también pueden cambiar el tamaño del texto en la pantalla haciendo zoom para verlo más grande. También pueden cambiar la orientación de su pantalla de paisaje a retrato mientras utiliza su programa. Si corrige la posición de los elementos en la pantalla, puede que se vea bien para un dispositivo en particular, pero que se vea horrendo en otro. Por esta razón, debe usar las funciones de diseño automático como `StackPanel` para que posicione dinámicamente las cosas por usted. Esto hace que su programa tenga muchas menos probabilidades de tener problemas de visualización.

Elementos y objetos XAML

Desde el punto de vista de la programación, cada uno de los elementos XAML presentados en una pantalla es en realidad un objeto de software. Los objetos son una gran forma de representar los elementos con los que requerimos trabajar, pero también sirven para representar otras cosas, como los elementos en una pantalla. Si piensa sobre esto, un cuadro que muestre un texto en pantalla tendrá propiedades como la posición en la pantalla, el color del texto, el propio texto en sí, etc.

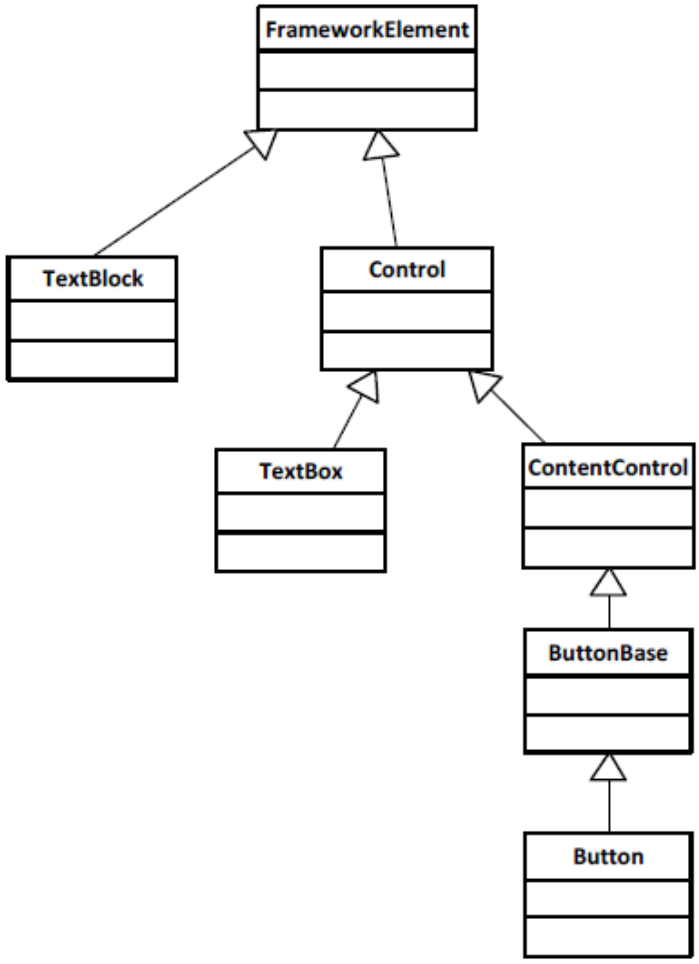
Usted está (o debería estar) familiarizado con el proceso de compilación, donde el código fuente (es decir, lo que tecleamos) para un programa de C# se convierte en un conjunto de instrucciones de bajo nivel que serán ejecutadas por la computadora. Cuando se compila un programa que utiliza la interfaz de usuario XAML, el sistema también "compila" la descripción XAML para crear un conjunto de objetos C#, cada uno de los cuales representa un elemento de interfaz de usuario. Hay tres tipos diferentes de elementos en la máquina sumadora:

1. `TextBox` – permite al usuario introducir texto en el programa.

2. `TextBlock` – un bloque de texto que solo transmite información.

3. `Button` – un elemento que podemos presionar para causar eventos en nuestro programa.

Si pienso sobre esto, puede dividir las propiedades de cada uno de estos elementos en dos tipos, aquellos que todos los elementos necesitan tener, por ejemplo, la posición en la pantalla, y los que son específicos de ese tipo de elemento. Por ejemplo, únicamente un `TextBox` necesita registrar la posición del cursor donde se introduce el texto. Desde el punto de vista del diseño de software, este es un uso muy bueno para una jerarquía de clases.



Arriba puede ver parte de la jerarquía que los diseñadores de XAML crearon. La clase superior es denominada `FrameworkElement`. Esta contiene toda la información que es común a todos los controles en la pantalla. Cada una de las otras clases, es hija de ésta. Las clases hijas heredan todos los comportamientos y propiedades de sus clases padres, y añaden algunos otros propios. En realidad, el diseño es un poco más complejo que el mostrado arriba, la clase `FrameworkElement` es hija de una clase llamada `UIElement`, pero esto muestra los fundamentos existentes detrás de los controles.

Crear una jerarquía de clases como esta tiene muchas ventajas. Si queremos un tipo de cuadro de texto personalizado, podemos extender la clase `TextBox` y añadir las propiedades y comportamientos que necesitamos. En lo que respecta al software que implementa el sistema XAML, puede tratar todos los controles de la misma manera y luego pedir a cada control que se dibuje de manera apropiada para ese componente.

Nuestro programa manipulará los elementos como si fueran objetos de C#, aunque en realidad se definieron en un archivo de origen XAML. Esto funciona porque cuando se construye el programa,

el sistema XAML creará objetos que coincidan con los elementos descritos en el archivo de origen XAML.

Administrar Nombres de Elementos

Cuando queremos utilizar los elementos de la interfaz de usuario en nuestro programa, necesitamos una forma de referirnos a cada uno de ellos. Si echamos un vistazo al XAML que implementa nuestra máquina sumadora, podemos ver que a algunos de los componentes se les ha dado una propiedad de nombre:

```
<TextBox Name="primerNumeroTextBox" Width="100" Margin="0,10"
         TextAlignment="Center" /></TextBox>
```

Yo he establecido el nombre de esta caja de texto a `cajaDeTextoPrimerNumero`. Seguro que nunca adivinarías el nombre de la segunda caja de texto. Debe tener en cuenta que el nombre de una propiedad en este contexto, realmente va a establecer el nombre de la variable declarada dentro del programa de la máquina sumadora. En otras palabras, como resultado de lo que he hecho anteriormente, ahora existirá la siguiente instrucción en alguna parte de mi programa:

```
TextBox primerNumeroTextBox;
```

Estas declaraciones se crean automáticamente cuando se construye el programa y, por lo tanto, no es necesario preocuparse dónde se encuentra realizada la declaración anterior. Nosotros solamente tenemos que recordar que así es como funciona el programa. También debe tener en cuenta que no todos los elementos en mi interfaz de usuario tienen nombres, no tiene sentido otorgar un nombre al `TextBlock` que contiene el carácter "+", ya que nunca necesitaré interactuar con éste cuando se ejecute el programa.

Propiedades de Elementos

Una vez que hemos otorgado un nombre propio a nuestra variable `TextBox`, podemos pasar a establecer todas las propiedades que son necesarias para esta aplicación. También podemos cambiar un montón de propiedades de la caja de texto, incluyendo el ancho de la caja, el margen (que establece la posición), etc.

Cuando nosotros hablamos de las "propiedades" de los elementos de Silverlight en la página (por ejemplo, el texto mostrado en un `TextBox`), en realidad estamos hablando de valores de propiedad en la clase que implementa el `TextBox`. En otras palabras, cuando un programa contiene una instrucción como:

```
resultadoTextBlock.Text = "0";
```

- Hará que se ejecute un método `Set` dentro del objeto `resultadoTextBox`, que establece el texto del `TextBlock` al valor apropiado.

Usted podría estar haciéndose la siguiente pregunta, "¿Por qué nosotros debemos utilizar propiedades en los elementos XAML?" Es más sensato hacerlo en una cuenta bancaria donde yo quiero tener la posibilidad de proteger los datos contenidos dentro de mis objetos, pero en una página XAML, donde yo puedo establecer el texto que quiera dentro de un `TextBlock`, parece que no tiene sentido establecer el valor entrante. De hecho, al ejecutar este código el proceso de establecimiento se ralentizará. Entonces, al hacer que el valor `Texto` sea una cadena pública, podríamos hacer que el programa sea más pequeño y rápido, lo que es todavía mejor. ¿Cierto?

Bueno, hasta cierto punto sí. Exceptuando que no está teniendo en cuenta que cuando cambiamos el texto de un `TextBlock`, también necesitamos que el texto que aparece en la página XAML, se actualice. Ya que así es como nuestra máquina sumadora mostrará el resultado. Si un programa simplemente cambia el valor de un miembro de datos, no habría forma de que el sistema XAML supiera que el mensaje mostrado en la pantalla debe actualizarse.

Sin embargo, si el miembro `Text` es una propiedad, cuando un programa lo actualice, el comportamiento establecido en el `TextBlock` se ejecutará. El código establecido en el valor de comportamiento puede actualizar el valor almacenado del campo de texto, y también puede desencadenar una actualización de la pantalla para hacer que el nuevo valor sea visible. Las propiedades proporcionan un medio a través del cual un objeto, puede tomar el control cuando se cambia un valor dentro de éste, lo que es extremadamente importante. La simple instrucción:

```
resultadoTextBlock.Text = "0";
```

- puede hacer que cientos de instrucciones C# se ejecuten, cuando el nuevo valor sea almacenado en el `TextBlock`, y desencadenar una actualización de pantalla para que se visualicen los cambios realizados.

Diseño de página con XAML

XAML resulta ser muy útil. Una vez que se familiariza con la forma que se utiliza para describir los componentes, resulta mucho más rápido agregar elementos a una página y posicionarlos editando el texto en el archivo XAML, que hacerlo moviéndose entre los valores de las propiedades. Yo encuentro esto particularmente útil, cuando quiero disponer de una gran cantidad de elementos similares en la pantalla. Visual Studio conoce la sintaxis utilizada para describir cada tipo de elemento y le proporcionará el soporte de Intellisense en tiempo de escritura.

Si lee la especificación XAML, encontrará que puede darles a los elementos propiedades gráficas que los hagan transparentes, agreguen imágenes a sus fondos e incluso animarlos a través de la pantalla. En este punto, nosotros estaríamos saliendo del campo de la programación, y entrando en el ámbito del diseño gráfico. Le deseo mucha suerte.

5.7.2 Creación de una Aplicación XAML completa

Ahora que sabemos que los elementos en la pantalla son los que constituyen los objetos del software, lo siguiente que necesitamos saber es cómo obtener el control de estos objetos y hacer que hagan cosas útiles para nosotros en nuestra aplicación. Para hacer esto nosotros tenemos que añadir algún código en C#, que realice el cálculo que la máquina sumadora requiere.

Cada vez que Visual Studio cree un archivo XAML que describa una página de pantalla, también creará un archivo subyacente con el mismo nombre escrito en C#. En este archivo es donde nosotros podemos establecer el código que hará que nuestra aplicación funcione. Si echa un vistazo al archivo `MainPage.xaml.cs`, verá que en realidad no contiene mucho código:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace ejemplo
{
   /// <summary>
   /// Lógica de interacción para MainWindow.xaml
   /// <summary>
   public partial class MainWindow : Window
   {
      public MainWindow()
      {
         InitializeComponent();
      }
   }
}
```

La mayor parte del código del archivo está formado por declaraciones que permiten que nuestro programa haga uso directo de las clases, sin tener que especificar el nombre completo de cada uno de ellas. Por ejemplo, en lugar de tener que escribir `System.Windows.Controls.Button` nosotros podemos escribir simplemente `Button`, porque el archivo cuenta con la línea `using System.Windows.Controls`.

Los únicos métodos en el programa son el constructor de la clase `MainWindow`. Como nosotros ya sabemos, el constructor de una clase es invocado cuando se crea una instancia de la clase. Todo lo que hace el constructor es llamar al método `InitializeComponent`. Si echa un vistazo al interior de ese método, encontrará el código que crea realmente las instancias de los elementos de visualización. Visual Studio crea automáticamente este código basándose en el XAML que describe su página. Es importante que deje esta llamada tal como está y no cambie el contenido del método, ya que es muy probable que esto afecte a su programa.

Tenga en cuenta que en este momento estamos en la "sala de máquinas" de XAML. Esto se lo estoy diciendo para que entienda que en realidad no existe ningún tipo de magia aquí. Si los únicos programas C# que ha visto hasta ahora comienzan con una llamada al método `Main`, entonces es importante que comprenda que no hay nada particularmente especial en un archivo XAML. Existe un método `Main` en la base de una aplicación XAML; que se encarga de iniciar el proceso de construcción de los componentes y de colocarlos en la pantalla para que el usuario interactúe con ellos.

El aspecto positivo en lo que a nosotros respecta es que no necesitamos preocuparnos sobre cómo se crean y se muestran estos objetos, solamente utilizaremos las herramientas de alto nivel o el código XAML fácilmente de entender, para diseñar y construir nuestra pantalla.

Crear la Aplicación

Ahora poder ver cómo crear la interfaz de usuario para nuestro programa destinado a sumar números. Si añadimos todos los componentes y, a continuación, iniciamos la aplicación incluso parece como si ya fuera del todo funcional. Nosotros podemos introducir los números que queremos sumar, e incluso hacer clic en el botón encargado de realizar la operación. Parece que tenemos muchos de los comportamientos que requerimos incorporados sin haber realizado demasiado esfuerzo, lo cual es algo bastante positivo. Sin embargo, nosotros necesitamos añadir alguna lógica de negocio propia, para hacer que el programa nos ofrezca la respuesta que requerimos y la muestre en pantalla.

Calcular el Resultado

En este momento nuestro programa muestra la apariencia que hemos especificado, pero realmente no hace nada. Nosotros necesitamos crear el código que realice el cálculo requerido y muestre el resultado en pantalla. Uno como este.

```
private void calcularResultado()
{
    float v1 = float.Parse(primerNumeroTextBox.Text);
    float v2 = float.Parse(segundoNumeroTextBox.Text);

    float resultado = v1 + v2;
```

```
        resultadoTextBlock.Text = resultado.ToString();
    }
```

Los objetos `TextBox` exponen una propiedad denominada `Text`. Esta puede ser leída o escrita. Al establecer un valor en la propiedad `Text`, cambiará el texto que se muestra en la caja de texto. La lectura de la propiedad `Text` permite que nuestro programa lea lo que se ha escrito en la caja de texto.

El texto es proporcionado como una cadena, la cual debe ser convertida en un valor numérico si nuestro programa va a hacer sumas. Usted ya ha visto cómo funciona el método `Parse` anteriormente. Este método toma una cadena y devuelve el número especificado en la cadena. Cada uno de los tipos numéricos (`int`, `float`, `double` etc.) tiene un comportamiento `Parse` que tomará una cadena y devolverá el valor numérico que ésta especifica. La máquina sumadora que estamos creando puede trabajar con números de punto flotante, por lo que el método analiza el texto en cada uno de los textboxes de entrada y, a continuación, ofrece el resultado de la suma de ambos.

Finalmente, el método toma el número del resultado de la operación, lo convierte en una cadena de texto, y a continuación, establece el texto del `resultadoTextBlock` en esta cadena. `ToString` es el método inverso de `Parse`, el "anti-parse" si lo prefiere llamar así. Este proporciona el texto que describe el contenido de un objeto. En el caso del tipo `float`, este es el texto que describe ese valor.

Ahora que nuestro código ya ofrece la respuesta, solo tenemos que encontrar la forma de que se ejecute cuando el usuario haga clic en el botón Igual a.

Eventos y Programas

Si ha realizado algún tipo de programación basada en formularios, seguro que sabrá todo sobre los eventos. Si nunca ha realizado este tipo de programación, no se preocupe, nosotros ahora vamos a exponer un buen ejemplo de un caso donde necesitamos utilizarlos, y va a comprobar que no son tan aterradores como parecen. En los viejos tiempos, antes de que se crearan las interfaces gráficas de usuario y los ratones, un programa generalmente se ejecutaba, funcionaba durante un tiempo y luego finalizaba.

Pero nosotros ahora tenemos interfaces de usuario complejas y llenas de botones y otros elementos con los que el usuario puede interactuar. El procesador de textos que estoy usando en este momento para escribir este texto, expone cientos de funciones diferentes a través de los botones de la pantalla y los elementos del menú. En esta situación, sería muy difícil escribir un programa que verificara cada elemento, para ver si el usuario ha intentado utilizarlo. En su lugar, el sistema espera que estos elementos que se muestran en pantalla, generen un evento cuando requieren atención. Los maestros hacen esto todo el tiempo. Ellos no van niño por niño, preguntando si conocen la respuesta. En lugar de esto, ellos piden a los niños que levanten la mano. La "mano levantada" es tratado como un evento al que el profesor responderá.

El uso de eventos como éste hace que el diseño de software sea mucho más fácil. Nuestro programa no tiene que verificar cada elemento de la pantalla para comprobar si el usuario ha realizado alguna con acción con éste, sino que simplemente atiende a los eventos que requieren su atención.

Para hacer que los eventos funcionen, un lenguaje de programación necesita una manera de expresar una referencia a un método de un objeto. C# proporciona el tipo `delegate (delegado)` que hace exactamente eso. Puede crear un tipo de delegado que pueda hacer referencia a un tipo concreto de método y, a continuación, crear instancias de dicho delegado que hagan referencia a un método en un objeto. Los delegados son muy poderosos, pero pueden ser un poco difíciles de entender. Afortunadamente, no tenemos que preocuparnos sobre cómo trabajan los delegados en este momento; porque podemos hacer que Silverlight y Visual Studio hagan todo el trabajo duro por nosotros.

Eventos en XAML

En C#, un evento es entregado a un objeto por medio de una llamada a un método de ese objeto. A este respecto, puede considerar que un evento y un mensaje son la misma cosa. Desde el punto de vista de nuestra máquina sumadora, nos gustaría tener un método particular que sea invocado cuando el botón "Igual a" es presionado por el usuario. Este es el único evento en el que nosotros estamos interesados. Cuando se desencadene el evento, nosotros queremos que éste realice una llamada al método `calcularResultado` que vimos anteriormente. Si echamos un vistazo al archivo XAML anterior, para comprobar el funcionamiento del botón, podemos comprobar a través de su código cómo se logra hacer esto.

```
<Button Content="Igual a" Name="botonIgualA"
  HorizontalAlignment="Center" Margin="0,10"
  Click= ="botonIgualA_Click"></Button>
```

El elemento botón tiene un nombre, `botonIgualA`, y una propiedad denominada `Click`. Esta propiedad tiene establecido el valor `botonIgualA_Click`. Este es el enlace establecido entre el XAML (que es el lenguaje *declarativo* que describe la página en la pantalla) y C# (que es el lenguaje de programación que realmente se encarga de hacer las cosas por nosotros).

Para que esto funcione, la página que contiene el elemento de visualización `botonIgualA` **debe** contener un método llamado `botonIgualA_Click`. De lo contrario, el programa no compilará correctamente. Este es el método que proporciona el comportamiento para el botón `IgualA`:

```
private void botonIgualA_Click(object sender, RoutedEventArgs e)
{
    calcularResultado();
}
```

El controlador de eventos debe tener la firma correcta para que se le puedan proporcionar los parámetros que describen el evento.

Si se está preguntando cómo funciona, este método realmente utiliza *delegados* para gestionar los eventos que se pasan desde el gestor de ventanas de Windows al programa que se encuentra en ejecución. Como ha visto anteriormente, un delegado es un objeto que representa un método en una instancia de una clase. Si piensa en ello; eso es exactamente lo que necesitamos aquí. Nosotros necesitamos tener una manera de comunicarle al administrador de sistema de ventanas de Windows, que cuando se haga clic en el botón que calcula el resultado de la suma, se invoque este método. Un objeto delegado es perfecto para esta tarea. El elemento de visualización `Button`, contiene una propiedad llamada `Click`. Un programa puede añadir objetos delegados a esta propiedad para que cuando el gestor de ventanas detecte que se ha hecho clic en el botón, éste pueda llamar a ese método. En otras palabras, en algún lugar del código generado para el XAML anterior, se encuentra la instrucción:

```
botonIgualA.Click +=new RoutedEventHandler(botonIgualA_Click);
```

El operador `+=` es un poco confuso en este caso, pero lo que este realmente significa es “añade un objeto delegado a la lista de aquellos eventos que serán invocados cuando el usuario haga clic en el botón”. El delegado es de tipo `RoutedEventHandler` y cuando una instancia del delegado es construida, se le proporciona el identificador del método al que va a hacer referencia. Si un programa alguna vez necesita desconectar un controlador de un evento, puede utilizar el operadorador `-=` para hacerlo.

Aplicaciones Dirigidas por Eventos

Tenga en cuenta que esto presenta un gran contraste con la forma en que los programas eran ejecutados en "los buenos viejos tiempos”. Antes, un programa se ejecutaba cuando se llamaba al método `Main`, y luego se detenía cuando se alcanzaba el final del método. Ahora nos encontramos en una situación donde realmente nunca llamamos al método para ejecutar el programa. El método es invocado para nosotros como respuesta a una acción realizada por el usuario. Lo que realmente sucede es que el controlador de eventos en el botón crea un hilo que llama al método del botón en nuestro programa.

Si tenemos una aplicación que utiliza muchos botones, necesitamos asegurarnos de que el programa es capaz de lidiar con los imprevistos que se pueden producir, si se hace clic en los botones en un orden ilógico. Por ejemplo, si un usuario presiona el botón Imprimir antes de haber cargado un documento, es importante que el programa no se cuelgue en este punto.

Lidiando con errores

La solución que hemos creado funciona correctamente, y proporciona una máquina sumadora de dos dígitos numéricos. Sin embargo, no puede decirse que el programa presente una interfaz muy amigable.

Por ejemplo, podría introducir los números tal y cómo vemos en la imagen de arriba. Nosotros sabemos que esto causará problemas con el método `Parse`, que lanzará una excepción y detendrá el programa. Nosotros también sabemos que podemos capturar excepciones y gestionarlas de manera adecuada. En el caso del programa anterior, la mejor forma de tratar con este tipo de errores es mostrar una ventana modal con un mensaje:

```
private void calcularResultado()
{
    try
    {
        float v1 = float.Parse(primerNumeroTextBox.Text);
        float v2 = float.Parse(segundoNumeroTextBox.Text);

        float resultado = v1 + v2;

        resultadoTextBlock.Text = resultado.ToString();
    }
    catch
    {
        MessageBox.Show("Número no válido", "Máquina sumadora");
    }
}
```

Código de Ejemplo 57 Máquina Sumadora

Esta versión del método `calcularResultado` captura la excepción lanzada por `Parse`, y a continuación, utiliza la clase `MessageBox` para mostrar un mensaje de error.

La clase `MessageBox` es una clase estática que forma parte de la biblioteca XAML. Existen varias versions del método `Show` para que pueda mostrar diferentes versiones de esta ventana modal. La versión que yo he utilizado, solamente acepta dos cadenas de texto. La primera de ellas, se utiliza para mostrar el mensaje de información destinado al usuario. La segunda de ellas, permite establecer el título de la ventana. El usuario puede cerrar la ventana modal, hacienda clic en el botón OK.

Tenga en cuenta que este método no sirve de mucho, ya que no indica cuál de los dos números introducidos provocó el error.

Punto del Programador: A los clientes les importa la interfaz de usuario

Si hay una parte del sistema en la que se garantiza que el cliente tiene una opinión sólida, esta es la de la interfaz de usuario. El diseño de ésta debe comenzar desde el inicio del proyecto y ser refinada a medida que avanza. Nunca jamás asuma que sabe que la interfaz de usuario debería funcionar de una manera particular. Yo he pasado más tiempo rediseñando el código de la interfaz de usuario que en cualquier otra parte del sistema. La buena noticia es que con Visual Studio es muy fácil crear prototipos bastante realistas del front-end de un sistema. Estos prototipos se pueden mostrar al cliente (incluso conseguir que los firme) para establecer se que está haciendo el trabajo correctamente.

5.8 Depuración de Programas

Algunas personas han nacido para depurar. No todo el mundo es bueno depurando, algunos siempre serán mejores que otros. Una vez dicho esto, existen técnicas que puede usar para facilitar el proceso, y vamos a explorar algunas de ellas aquí. La buena noticia es que, si realiza un desarrollo controlado por pruebas, el número de fallos que los usuarios deben encontrar debe ser lo más bajo posible, pero, aun así, habrá algunas cosas que deba corregir.

Por cierto, la razón por la que los errores que surgen en los programas sean llamados bugs (proveniente del término insecto, en inglés), se debe a que el originario fue causado por un insecto real que se quedó atascado en un contacto de la computadora, causando que fallara. Sin embargo, esta no es la causa por la que aparecen la mayoría de los errores. Es muy raro ver que su programa falle porque el hardware está defectuoso. He estado programando durante muchos años y solo he visto que ésta haya sido la causa en un número ínfimo de ocasiones. Lo triste es que la mayor parte de los bugs que se encuentran en los programas han sido introducidos por los propios programadores. En otras palabras, la depuración de programas es el proceso de identificar y corregir errores de programación.

5.8.1 Reportar fallos

Ya hemos visto que un fallo es algo que el usuario ve como resultado de un error en su programa.

Los fallos son descubiertos a través del proceso de testeo o por los usuarios. Si un programa falla como parte de un testeo, los pasos tomados para manifestarlo serán registrados. Sin embargo, Si un usuario reporta un fallo, la prueba que demuestran que ese fallo existe puede ser anecdótica, es decir, no le será proporcionada una secuencia de pasos que provocan la aparición del fallo, simplemente le indicarán "Hay un error en la rutina de impresión".

Existe un fuerte argumento para ignorar el reporte de un fallo por parte de un usuario, si no le ha dado una secuencia de pasos a seguir para provocarlo. Sin embargo, este enfoque no es apreciado por los usuarios. Es importante hacer resaltar a los usuarios que cualquier informe de fallos solamente será tomado en cuenta si está bien documentado.

Punto del Programador: Diseñe el Proceso de Reporte de Fallos

Como casi todas las demás cosas que hemos mencionado hasta este instante, la forma en que administra el reporte de fallos de su programa, debe ser considerada al inicio del proyecto. En otras palabras, debe establecer un proceso para hacer frente a los fallos que se reporten. En un desarrollo a gran escala, los informes de fallos son administrados, asignados a los programadores y monitorizados con atención. El número de fallos que son reportados, y el tiempo que se toma para solventarlos, es un dato valioso para reconocer la calidad del trabajo que está realizando. Además, si formaliza (o quizás incluso automatiza) el proceso de notificación de fallos, puede asegurarse de estar obteniendo la máxima información posible sobre el problema.

Los dos tipos de Fallos existentes

Los fallos se dividen en dos tipos, los que siempre suceden y los que a veces suceden.
Los fallos que siempre suceden son más fáciles de resolver, puede realizar la secuencia que siempre causa que el error se manifieste y luego usar las técnicas adecuadas para ponerlo al descubierto (ver más adelante).

Los fallos que suceden a veces son un dolor. Lo que esto significa es que no se tiene una secuencia definida de eventos que tenga como consecuencia que el sistema falle. Esto no significa que no exista realmente una secuencia (a menos que esté siendo inducido por algún tipo de fallo de hardware – lo cual es bastante extraño) sino que usted todavía no la ha descubierto.

Un ejemplo sería una función de impresión que a veces se cuelga y otras veces funciona sin problemas. Podría realizar un seguimiento de este fallo, teniendo en cuenta si falla dependiendo de número de páginas impresas, o la cantidad de texto existente en la página, o el tamaño del documento que se está editando cuando se solicita la impresión.

La aparición de un fallo puede suceder después de que el error se haya producido, por ejemplo, un programa puede fallar cuando se elimina un elemento de una base de datos, pero el error puede estar en la rutina que almacenó el elemento o en un código que sobrescribió la memoria donde se encuentra localizada la base de datos.

Un fallo puede cambiar o desaparecer cuando se realizan cambios en el propio programa; los errores que sobrescriben la memoria corromperán diferentes partes del programa si cambia el diseño del código. Esto puede llevar al tipo de fallo más molesto, donde usted introduce instrucciones de impresión adicionales para obtener más información sobre el problema, ¡y el fallo desaparece!

Si sospecha que su programa está presentando un error de este tipo, la primera prueba que debe realizar es cambiar el código y la forma en que se encuentran distribuidos los datos, y volver a

ejecutar el programa. Si el fallo cambia de naturaleza, esto es un síntoma que evidencia problemas internos del programa o que los datos están dañados.

5.8.2 Acabar con los bugs

Puede dividir los fallos en otras dos categorías, cuando el programa se bloquea (es decir, deja de responder por completo) o cuando hace algo incorrecto. Por sorprendente que parezca, los bloqueos son frecuentemente más fáciles de solventar. Estos usualmente apuntan a:

- una condición que no ha sido tratada correctamente (asegúrese que todas las instrucciones de selección, tengan un controlador predeterminado que haga algo razonable, y que utiliza técnicas de programación defensiva cuando los valores se transfieren entre los módulos).

- programas que se quedan atrapados en bucles, construcciones `do – while` que nunca se cumplen, bucles `for` que contienen código que cambia la variable de control, métodos que se llama así mismos por error de manera recursiva provocando el desbordamiento de la pila.

- una excepción que no ha sido capturada correctamente

Si su programa hace algo incorrecto, esto puede ser difícil de encontrar. Busque:

- uso de miembros de clases no inicializados.

- errores tipográficos (busque nombres de variables escritas de manera equivocada, operadores de comparación inadecuados, comentarios de código no cerrados correctamente).

- errores lógicos (busque fallos en las secuencias de instrucciones, terminaciones de bucles no válidas, condiciones lógicas mal construidas).

Como dije anteriormente, algunas personas son especialmente buenas a la hora de encontrar bugs. Aquí tiene algunos consejos:

No hagas suposiciones. Si supone que "la única manera de que esto pudiera llegar aquí, es a través de esta secuencia" o "no hay manera de que este fragmento de código pueda cumplirse", puede que se equivoque. En lugar de hacer suposiciones de este tipo, añada código adicional para probar que lo que piensa está realmente sucediendo.

Hable con el limón

Explique el problema a otro, incluso – ¡si es solo un gato, un pato de goma o un simple limón! El acto de explicar un problema puede llevarle a deducir la respuesta. Lo mejor es que la persona con la que esté hablando sea muy escéptica con su testimonio.

Revise todos los factores posibles (aunque aparentemente no deban estar relacionados) en la aparición del bug. Esto es particularmente importante si el bug es intermitente. Si el bug aparece los viernes por la tarde en su sistema UNIX, averigüe si otro departamento utiliza la máquina para ejecutar un proceso de cálculo de nóminas en este momento que está ocupando todo el espacio de temporal de almacenamiento del disco duro, o el departamento de carga ha puesto en funcionamiento el polipasto emitiendo un montón de ruido a la red.

Deje el problema aparcado por un tiempo. Dedíquese a hacer algo diferente y comprobará que la respuesta aparece ante usted por sí sola. De manera alternativa, encontrará la respuesta tan pronto como vuelva al problema.

Recuerde que, aunque el bug es de curso/naturaleza imposible, este se está presentando. Esto significa que o bien lo imposible está sucediendo, ¡o su suposición de lo imposible es errónea!

¿Puede volver a un estado previo del proyecto donde el bug no estaba presente, y comprobar los cambios realizados desde entonces? Si el sistema está fallando como resultado de un cambio o, el cielo no lo quiera, un bug corregido; pruebe a volver a un punto anterior donde el bug no estaba presente, y luego introduzca los cambios hasta que aparezca el error. De manera alternativa, examine con detalle cómo la introducción de cada nueva función afecta a otros módulos en el sistema. Un buen Sistema de Control de Código Fuente, es muy valioso en este tipo de situaciones, ya que le informa exactamente de los cambios realizados en el código fuente de su programa de una versión a otra.

Una cosa que debo dejar claro en este punto es que el proceso de depuración es el de corregir fallos en una solución que debería funcionar. En otras palabras, usted debe conocer cómo se supone que el programa debe funcionar, antes de tratar de solucionar problemas que realmente no existen. Me conmueve ver a algunos programadores estableciendo otro bucle, o cambiando la forma en la que las condiciones del bucle operan para "ver si esto hace que el programa funcione". Tales esfuerzos siempre están condenados al fracaso, al igual que tampoco va a servir de nada tirar un montón de componentes electrónicos contra la pared y esperar a que cuando todas terminen de caer al suelo, se haya formado un reproductor de DVD.

Borrón y cuenta nueva

En algunos proyectos, es posible que el esfuerzo involucrado en empezar de nuevo, sea menor que tratar de averiguar la causa que está quebrantando la solución que ha creado. Si ya se ha tomado un descanso, le ha explicado el código a un amigo, y ha comprobado las hipótesis entonces quizás, y solamente quizás esta sea la mejor manera de avanzar.

Punto del Programador: Corregir Bugs Provocan Bugs

La causa principal de que aparezcan bugs es probablemente el proceso de corrección de bugs. Esto se debe a que cuando los programadores modifican el código del programa para tratar de hacerlo funcionar, el cambio de código hace a menudo quebrantar otras partes del sistema. He encontrado estadísticas que indican que con frecuencia se suele producir un "dos por uno", es decir, por cada bug corregido se presentan dos bugs completamente nuevos. La única forma de evitar esto es asegurarse de que el proceso de pruebas (que ha creado como una serie de pruebas unitarias), se pueda ejecutar automáticamente después de aplicar la corrección. Esto al menos asegura de que la corrección que ha realizado no ha quebrantado nada importante.

5.8.3 Crear un Software perfecto

No existe el software perfecto. Una de las reglas con las que trabajo es que "cualquier programa útil tendrá errores". Dicho de otra forma, Yo puedo desarrollar programas que puedo garantizar que no contendrán errores. Sin embargo, estos programas serán muy pequeños y, por lo tanto, no harán demasiadas cosas. Tan pronto como yo empiece a crear programas útiles, con entradas, salidas y comportamientos, comenzaré a introducir bugs.

Esto no significa que cada programa que escriba no sirva para nada, solamente que no será perfecto. Al considerar los fallos, también debe considerar su impacto. Parte del trabajo en un desarrollo de un gestor de proyecto, es decidir cuándo un producto está lo suficientemente maduro como para ponerlo a la venta, y si un error en el código es o no una “barrera, freno o traba” (en la terminología anglosajona se suele utilizar el término “bug stopper”) para ello.

Este tipo de fallos hacen que el programa no sea vendible. Si el programa se bloquea cada tres veces que lo ejecuta, o a veces destruye los datos almacenados en la computadora host, estos son probablemente indicios asociados al comportamiento de este tipo de bugs. Pero si el programa hace cosas como imprimir siempre dos veces la primera página de un documento al utilizar el idioma chino en un modelo concreto de impresora láser, esto podría considerarse como un problema con el que la mayoría de los usuarios podrían convivir.

Esto significa que usted debe evaluar el impacto de los fallos que se le reporten, priorizarlos y administrar como se tratan. Por supuesto, es importante que tenga en cuenta y sea consciente del ámbito para el que está destinado el desarrollo que está realizando. Por ejemplo, un error en un videojuego es mucho menos problemático, que un error en un sistema de control de tráfico aéreo.

La clave para hacer que el software sea lo más perfecto posible, es asegurarse de que comprende bien el problema que está resolviendo, que sabe cómo resolverlo antes de comenzar a escribir el código, y que gestiona cuidadosamente el proceso de producción del código.

Lea algunos de los textos recomendados al final de este documento, para obtener más información sobre este aspecto de la programación.

5.9 ¿El final?

Esto no es todo lo que necesita saber para ser programador. Ni siquiera es todo lo que necesita saber para ser un programador de C#. Sin embargo, este es un buen comienzo, aunque falten algunas cosas en este texto que no han sido examinadas, porque llevaría demasiado tiempo hacerlo. Usted debe revisar los siguientes temas, si quiere convertirse en un gran programador de C#:

- serialización
- atributos
- reflexión
- redes

5.9.1 Desarrollo continuo

Un buen programador tiene una política deliberada de revisar constantemente su experiencia y aprender cosas nuevas. Si se toma en serio este negocio, debería leer al menos un libro sobre la materia siempre que pueda. Yo he estado programando desde que tengo uso de razón, pero nunca he dejado de aprender sobre la materia. Y nunca dejaré de programar, leer libros sobre programación y revisar el código de otras personas.

5.9.2 Lecturas complementarias

Code Complete Second Edition: Steve McConnell

Publicado por Microsoft: ISBN 0-7356-1967-0

En realidad, no es un libro sobre C#. Es más, un libro sobre todo lo demás. Este cubre un amplío aspecto de técnicas de programación y de ingeniería de software desde la perspectiva de la "construcción de software". Si tiene la seria intención de convertirse en programador profesional, deber leer/poseer este libro.

How to be a programmer

Este sitio web también vale la pena leerlo, ya que cubre muy bien los comportamientos a los que se enfrenta un programador en su vida diaria:

http://samizdat.mines.edu/howto/HowToBeAProgrammer.html

6 Glosario de Términos

Abstracto (Abstract)

Algo que es abstracto no tiene una existencia "propia" como tal. Al escribir programas, nosotros utilizamos esta palabra para indicar "una descripción idealizada de algo". En el caso del diseño de componentes, una clase abstracta contiene descripciones de elementos que recrean un comportamiento común, pero sin especificar cómo lo hacen. En términos de C#, una clase es abstracta si está marcada como tal o si contiene uno o más métodos que se marcan como abstractos.

No se puede crear una instancia de una clase abstracta, pero se puede utilizar como base o plantilla para una clase concreta. Por ejemplo, podemos decidir que necesitamos muchos tipos diferentes de recibos en nuestro sistema de procesamiento de transacciones: recibo de efectivo, recibo de cheque, recibo de mayorista, etc. No sabemos cómo funcionará cada recibo en particular, pero sí sabemos los comportamientos que debe tener para convertirlo en un recibo.

Nosotros podemos, por tanto, crear una clase `Recibo` abstracta que sirva como base para las otras clases concretas comunes. Cada clase de recibo "real" se crea extendiendo de la clase padre, la clase abstracta. Lo que significa que ésta es un miembro de la familia de recibos (es decir, puede ser tratado como un recibo) pero trabaja de una manera particular.

Acomplamiento

Si una clase es *dependiente* de otra, se dice que las dos clases están *acopladas*. En términos generales, un programador debe esforzarse por tener el menor acoplamiento posible en sus diseños, ya que esto dificulta la actualización del sistema.

El acoplamiento es examinado a menudo junto con la cohesión, ya que usted como programador, debería tener como objetivo que las clases de su sistema tengan una alta cohesión y un mínimo acoplamiento.

Anulación (Override)

A veces es posible que quiera crear una versión más especifica de una clase existente. Esto puede implicar el proporcionar versiones actualizadas de métodos en la clase. Puede hacer esto, creando una clase hija que extienda del padre, y a continuación, reemplazando los métodos que necesitan ser modificados. Cuando se invoca al método a través de las instancias de la clase hija, se llama al nuevo método, no al método del padre anulado/invalidado. Puede utilizar la palabra reservada `base`, para acceder al método anulado/invalidado si es necesario.

Archivo de código fuente

Usted elabora un archivo de código fuente utilizando un editor de texto. Este es el texto que usted pasa al compilador para producir un archivo de programa ejecutable.

Base

`base` es una palabra reservada de C# que tiene diferentes significados según el contexto en el que se encuentre. Se utiliza en un constructor de una clase hija, para llamar al constructor de la clase padre. También se utiliza en los métodos de reemplazo, para llamar al método que se han anulado/invalidado.

Biblioteca

Una biblioteca es un conjunto de clases que son utilizadas por otros programas. La diferencia entre una biblioteca y un programa es que un archivo de biblioteca tendrá la extensión .dll (*dynamic link library*, biblioteca de vínculos dinámicos en español), y no contendrá un método principal.

Clase

Una clase es una colección de comportamientos (métodos) y datos (propiedades). Se puede usar para representar un objeto del mundo real en el programa (por ejemplo, una cuenta bancaria).

Código Máquina

El Código Máquina es el lenguaje que el procesador de la computadora realmente entiende. Este contiene un número de operaciones muy simples, por ejemplo, mover un elemento desde el procesador a la memoria, o sumar uno a un elemento del procesador. Cada gama particular de procesadores de computadora tiene su propio código específico de la máquina, lo que significa que el código de máquina escrito para un tipo de máquina no se puede usar fácilmente en otro.

Cohesión

Una clase contiene una alta cohesión si no depende/no se encuentra acoplada a otra clase.

Colección

La biblioteca C# tiene por objetivo reunir un conjunto de elementos que quiera almacenar, por ejemplo, todos los jugadores de un equipo de fútbol o todos los clientes de un banco. Una matriz es una colección ordenada de elementos colocados en filas y columnas. Otra forma de colección es una tabla hash que le permite encontrar fácilmente un elemento en particular en función de un valor clave. Una clase de colección admitirá la enumeración, lo que significa que le puede pedir que proporcione valores sucesivos a la construcción `foreach` de C#.

Siempre que quiera almacenar un número de elementos de forma conjunta, debe considerar hacerlo utilizando una clase de colección. Las clases de colección se pueden encontrar en el espacio de nombres `System.Collections`.

Compilador

Un compilador toma un archivo de código fuente y lo traduce a lenguaje máquina. El compilador finalmente, producirá un archivo ejecutable que se podrá ejecutar en una máquina. Escribir compiladores es un negocio especializado, en épocas anteriores solían desarrollarse en lenguaje ensamblador, pero ahora están construidos en lenguajes de más alto nivel (¡como por ejemplo C#!). La mayoría de los compiladores trabajan en varias fases. En la primera fase, el preprocesador, toma el archivo de código fuente que el usuario ha escrito e identifica todas las palabras reservadas, identificadores y símbolos, produciendo un flujo de programa que es suministrado al "analizador sintáctico" que se asegura que la fuente respeta la gramática del lenguaje de programación en uso. La fase final genera el código que produce el archivo ejecutable, que luego es ejecutado por el *host* o anfitrión.

Componente

Un componente es una clase que expone su comportamiento en forma de interfaz. Esto significa que, en lugar de pensar en términos de lo que es (por ejemplo, una `CuentaClienteInfantil`), se piensa en términos de lo que puede hacer (implementar la interfaz `ICuenta` para pagar y retirar dinero). Al crear un sistema, debe centrarse en los componentes y la forma en que interactúan. Sus interacciones se expresan en las interfaces vinculadas entre ellas.

Constructor

Un constructor es un método de una clase que se invoca cuando se crea una nueva instancia de la clase. Los programadores utilizan constructores para tomar el control para establecer los valores dentro de la clase. Si una clase es miembro de una jerarquía y la clase padre tiene un constructor,

es importante cuando cree la clase hija que se asegure de que se llama correctamente al constructor padre. De lo contrario, el programa no compilará.

Delegado

Un delegado es un tipo de referencia segura a un método. Un delegado es creado para una firma de método particular (por ejemplo, un método que acepta dos valores de tipo entero, y devuelve un valor de tipo flotante). Este puede a continuación ser enviado a un método de una clase cuyos parámetros coincidan con los de esa firma. Tenga en cuenta que la instancia de delegado contiene dos elementos, una referencia a la instancia/clase que contiene el método y una referencia al propio método en sí. El hecho de que un delegado sea un objeto significa que puede ser transmitido como cualquier otro.

Los delegados se utilizan para informar a los generadores de eventos (cosas como botones, temporizadores y similares) del método que debe invocarse cuando el evento que generan tiene lugar.

Dependencia

En general, demasiada dependencia en sus diseños es algo negativo. Existe una relación de dependencia entre dos clases, cuando una modificación en el código de una de esas clases tenga como consecuencia el tener que cambiar el código de la otra clase también. Por lo general, significa que no ha asignado adecuadamente la responsabilidad entre los objetos en su sistema y que dos objetos están compartiendo los mismos datos. Como ejemplo de esto, consulte el método `Cargar` de las clases `CuentaCliente` y `CuentaInfantil` en la página 250.

La dependencia es a menudo direccional. Por ejemplo, una clase de interfaz de usuario puede depender de una clase de objeto de negocio (si agrega nuevas propiedades al objeto de negocio, deberá actualizar la interfaz de usuario). Sin embargo, es poco probable que los cambios en la forma en que funciona la interfaz de usuario signifiquen que el objeto de negocio deba modificarse.

Espacio de nombres

Un espacio de nombre es un área dentro de la cual se encuentran almacenados un conjunto de nombres en el cual todos los nombres son únicos. Los espacios de nombres permiten reutilizar nombres. Un programador que crea un espacio de nombres puede utilizar cualquier nombre en ese espacio de nombres. Un nombre completo de un recurso está prefijado, por el espacio de nombres en el que se encuentra el nombre de dicho recurso. Un espacio de nombre puede contener otro espacio de nombre, lo que permite establecer jerarquías. Debe tener en cuenta que un espacio de nombres es puramente lógico, ya que no refleja en qué parte del sistema se encuentran físicamente

los elementos, simplemente proporciona los nombres que los identifican. C# proporciona la palabra clave `using` para permitir que los espacios de nombres sean "importados" en un programa.

Especificación de diseño funcional

El desarrollo de grandes sistemas de software sigue un camino particular, desde la reunión inicial hasta cuando se entrega el producto. La ruta precisa seguida depende de la naturaleza del trabajo y las técnicas que utiliza el desarrollador; sin embargo, todos los desarrollos deben comenzar con una descripción de lo que el sistema debe hacer. Esta parte del desarrollo denominada bajo el nombre de Especificación de diseño funcional o FDS es la más crucial de todo el proyecto.

Estático - Static

En el contexto de C# la palabra reservada `static` hace que un miembro de una clase forme parte de una clase, en lugar de formar parte de una instancia de la clase. Esto significa que no es necesario crear una instancia de una clase para hacer uso de un miembro estático. Dado que no hay ninguna variable de instancia, para tener acceso a los miembros de una clase estática, debe usar el nombre de la clase. Los miembros estáticos son útiles para crear miembros de clase que sean compartidos con todas las instancias, por ejemplo, los tipos de interés para todas las cuentas bancarias.

Estructura

Una estructura es una colección de elementos de datos. Las estructuras no se pasan a los métodos por referencia, siempre se pasan por valor. Las estructuras son útiles para mantener fragmentos de datos relacionados en unidades individuales. Estas no son tan flexibles como los objetos administrados por referencia, pero son más eficientes de usar ya que el acceso a los elementos de la estructura no requiere que se siga una referencia de la misma manera que para un objeto.

Evento

Un evento es un suceso externo al que su programa puede necesitar responder. Los eventos incluyen cosas como el movimiento del ratón, el presionar teclas, ventanas que se redimensionan, botones que se presionan, un temporizador que espera un intervalo de tiempo para desencadenar un tick, etc. Muchos programas actuales trabajan con eventos conectados a los métodos. Cuando se produce el evento, se invoca al método para entregarle la notificación. Los componentes de Windows hacen uso de los delegados (un delegado es una referencia de tipo segura a un método) para permitir que los generadores de eventos sean informados del método que se debe invocar cuando se produce el evento.

Excepción

Una excepción es un objeto que describe algo inesperado que acaba de suceder. Las excepciones forman parte de la manera en que un programa C# puede lidiar con los errores. Cuando un programa en ejecución llega a una situación donde no puede continuar (por ejemplo, un archivo que no se puede abrir o un valor de entrada sin lógica) puede desistir en el intento y "lanzar" una excepción:

```
throw new Exception("¡Oh cielos!");
```

El objeto `Exception` contiene una propiedad `Message` de tipo cadena que puede ser utilizada para describir lo que salió mal. En el ejemplo anterior, el mensaje es establecido a "¡Oh cielos!".

Si la excepción no es "capturada", el programa finalizará en ese punto. Puede hacer que un programa responda a las excepciones encerrando el código que arroja la excepción en una construcción `try – catch`. Cuando se lanza la excepción, el programa transfiere la ejecución al código existente en la cláusula `catch`:

```
try
{
    // Código que podría lanzar una excepción
}
catch (Exception e)
{
    // Código que captura la excepción
    // Código que responde cuando se produce una excepción
    // e es una referencia a la excepción que fue lanzada
}
finally
{
    // Código que será ejecutado
    // se haya lanzado o no una excepción
}
```

Una construcción `try – catch` puede contener también una cláusula `finally`, que contiene código que se ejecuta independientemente de que se haya lanzado la excepción o no.

Firma

Todo método en C# tiene una firma particular que le permite ser identificado de manera inequívoca en un programa. La firma es definida por el nombre del método, el tipo y el orden de los parámetros esperados por ese método:

`void Absurdo(int a, int b)` – tiene como firma el nombre Absurdo y dos parámetros de tipo entero.

`void Absurdo(float a, int b)` – tiene como firma el nombre Absurdo y un parámetro de tipo flotante seguido de un parámetro de tipo entero. Esto significa que el código:

```
Absurdo(1, 2);
```

- invocaría al primero método, mientras:

```
Absurdo(1.0f, 2);
```

- invocaría al segundo método.

Tenga en cuenta que el tipo del método no tiene ningún efecto sobre la firma.

Herencia

La herencia es el mecanismo por la que una clase hija derivada de una clase padre puede hacer uso de todos sus comportamientos y propiedades, permitiéndole además añadir su propio comportamiento o modificar el heredado. Para obtener más información, consulte la descripción del término jerarquía.

Identificador único global (GUID)

El identificador único global (en inglés denominado *globally unique identifier* - GUID) es un número pseudoaleatorio empleado en aplicaciones de software con la intención de que un componente de software sea único en el mundo. Los GUIDs son utilizados por elementos como las referencias de cuenta y las matrículas que deben ser únicas. La mayoría de los sistemas operativos y las bibliotecas de programadores proporcionan métodos para crear GUIDs.

Inmutable

Un objeto inmutable no puede ser cambiado. Si se intenta cambiar el contenido de un objeto inmutable, se crea un nuevo objeto con el contenido modificado y el "viejo" permanece en la memoria. La clase `string` es inmutable. Esta característica proporciona a las cadenas un comportamiento similar a los tipos valor, lo que hace que sean más fáciles de utilizar en los programas.

Interfaz

Una interfaz describe un grupo de comportamientos relacionados entre sí, que pueden pertenecer a cualquier clase o estructura. Las acciones son definidas a través de los métodos descritos en la interfaz. Una clase que implementa una interfaz debe contener código para cada uno de los métodos. Una clase que implementa una interfaz puede ser puramente referenciada en términos de esta interfaz. Las interfaces permiten crear componentes. Con las interfaces nos alejamos de considerar las clases en términos de lo que son, y empezamos a pensar en ellas en términos de lo que pueden hacer.

Jerarquía

Una jerarquía es creada cuando una clase padre es extendida por una clase hija para producir una nueva clase que hereda todas las capacidades que la clase padre tiene, y añadiendo o modificando nuevos comportamientos específicos requeridos por la clase hija. La acción de extender una clase hija produce un nivel adicional jerárquico. Las clases en la parte superior de la jerarquía deberían ser habitualmente más generales y posiblemente tener un mayor nivel de abstracción (por ejemplo, una `CuentaBancaria`), mientras que las clases en los niveles inferiores serán más específicas (por ejemplo, una `CuentaBancariaInfantil`).

Lenguaje declarativo

Un lenguaje declarativo informa al sistema de una computadora acerca de las cosas. Este no proporciona instrucciones que explican cómo realizar una acción; solo hace que el sistema tome conciencia del hecho de que algo existe y que tiene un conjunto particular de propiedades.

C# contiene declaraciones; así es como un programa puede crear variables.

```
int i;
```

Esta declaración crea una variable de tipo entero identificada por el nombre `i`. No obstante, un programa C # también contiene declaraciones que le dicen a la computadora cómo realizar una acción.

```
i = i + 1;
```

Esta declaración suma uno al valor almacenado en la variable `i`. La capacidad de poder realizar esta acción, hace que C# no sea un lenguaje declarativo. Sin embargo, el lenguaje de marcado XAML es declarativo. XAML fue diseñado para especificar el diseño de una página. Todo archivo de código fuente XAML, contiene descripciones de elementos.

```
<TextBox Name="primerNumeroTextBox" Text="0"
 VerticalAlignment="Top" Width="460"
 TextAlignment="Center">
</TextBox>
```

Este fragmento de código XAML describe un elemento `TextBox`, donde se especifica la anchura, el nombre del elemento y la alineación en pantalla. Sin embargo, no hace ni puede indicar al sistema qué hacer con el elemento, ni especificar ningún comportamiento que tenga. Esto se debe a que el lenguaje de marcado XAML no proporciona ninguna manera de expresar el comportamiento que debe tener un elemento; simplemente está ahí para comunicar al sistema cómo deben visualizarse los elementos en pantalla.

Llamada

Cuando quiere utilizar un método, usted llama a éste. Cuando un método es llamado (esta acción también puede ser denominada como *invocado*), la secuencia de ejecución se traslada a ese método, procesando la primera instrucción de su cuerpo. Cuando se alcanza el final del método, o bien la instrucción `return`, la secuencia de ejecución retorna a la instrucción inmediatamente posterior de la llamada al método.

Metadatos

Los metadatos son “datos acerca de datos”. Estos operan a todos los niveles. El hecho de que el valor de la edad sea almacenado en una variable de tipo entero es un metadato. El hecho de que ésta no pueda ser negativa es otro metadato. Los metadatos deben ser recopilados por el programador junto con la ayuda del cliente antes de crear un sistema.

Método

Un método es un bloque de código precedido por una *firma de método*. Un método tiene un nombre particular identificativo y puede devolver un valor de retorno. Este también podría aceptar un parámetro con el que trabajar. Los métodos se utilizan para dividir el código en fragmentos más pequeños, cada una de las cuales realiza una parte de la tarea. También se utilizan para poder lograr que el mismo trozo de cógido pueda ser utilizado en otras partes del programa, sin la necesidad de tener que tener código duplicado. Si un método es declarado público puede ser invocado desde el código de otras clases. Un objeto expone sus comportamientos, a través de un método público. Un mensaje es entregado a un objeto a través de una llamada de un método dentro de ese objeto.

Método Virtual

Un método es un miembro de una clase. Yo puedo llamar al método para realizar un trabajo. En ocaciones, es posible que quiera extender una clase para producir una clase hija, lo que me permite crear una versión más especializada de esa clase. En cuyo caso, podría interesarme reemplazar el método de la clase padre por uno nuevo en la clase hija. Para que esto suceda, el método de la clase padre debe ser marcado como `virtual`. Solo los métodos virtuales pueden ser anulados. Crear un método virtual, hace que se ralentice un poco el proceso de acceso a éste, ya que el programa debe buscar cualquier modificación del método antes de invocarlo. Esta es la razón por la que no todos los métodos se marcan como virtuales desde un principio.

Miembro

Un miembro de una clase es declarado dentro de esa clase. Este puede hacer algo (un método), o contener algunos datos (variable). Los métodos son a veces denominados como comportamientos. Los miembros de datos son a veces denominados propiedades.

Mutador

Un mutador es un método que es invocado para cambiar el valor de un miembro dentro de un objeto. El cambio debería ser comprobado antes de realizarse, ya que, si se recibe un valor no válido deberá ser rechazado de alguna manera. Esto es implementado en forma de un método público que suministra el nuevo valor y que puede retornar un código de error.

Pereza creativa

Me parece que algunos aspectos de la pereza son igualmente válidos para aplicarlos al campo de la programación. La reutilización de código, donde se intenta aprovechar un código ya existente, es un buen ejemplo de esto. Asegurarse de que la especificación es válida antes de hacer cualquier cosa, es otra forma de ahorrar tiempo de trabajo. No obstante, estructurar el diseño para que otro pueda hacer el trabajo requerido, es probablemente el mejor ejemplo que se puede dar de la pereza creativa.

Portable

Cuando se aplica a un programa informático, cuanto más portátil es algo, más fácil es portarlo a otro sistema. Las computadoras están fabricadas con diferentes tipos de procesadores y pueden utilizar distintos sistemas operativos por lo que solamente pueden ejecutar programas escritos específicamente para ellas. Una aplicación portable es aquella que puede ser portada a un nuevo

procesador o sistema operativo con relativa facilidad. Los lenguajes de alto nivel tienden a ser portables, el código máquina es mucho más difícil de portar.

Privado (Private)

Un miembro privado de una clase solamente es visible a nivel de código por métodos existentes dentro de esa clase. Es convencional hacer que los miembros de los datos de una clase sean privados, para que no puedan ser modificados por clases externas. El programador puede **proporcionar métodos o propiedades C# para administrar los valores** que pueden ser asignados **a los miembros privados**. **La única razón para no** marcar un miembro de dato como privado, es para evitar el impacto en el rendimiento que se produce al acceder a los datos del miembro a través de un método.

Propiedad

Una propiedad es un elemento de dato mantenido en un objeto. Un ejemplo de una propiedad de una clase `CuentaBancaria` sería el saldo de la cuenta. Otro sería el nombre del titular de la cuenta. El lenguaje C# tiene una construcción especial que facilita el manejo de propiedades a los programadores.

Protegido (Protected)

Un miembro protegido de una clase es visible para los métodos existentes en la clase y para los métodos en las clases que extienden de esta clase. Esta es una solución a medio camino entre el modificar privado (que no permite el acceso a los miembros a métodos fuera de esta clase) y el modificador público (todas las clases tienen acceso a los miembros). Este modificador, permite designar miembros visibles tanto en clases padres como en clases hijas.

Prueba unitaria

Una prueba unitaria es una pequeña prueba que expone un componente y se asegura que realiza una función particular correctamente. Las pruebas unitarias deben escribirse junto con el proceso de desarrollo para que se puedan aplicar al código inmediatamente después de que (o justo antes en el desarrollo guiado por puebas) el código haya sido escrito.

Público (Public)

Un miembro público de una clase es visible por los métodos fuera de la clase. Es convencional hacer públicos los miembros del método de una clase, para que puedan ser utilizados por otras clases. Un método público es el mecanismo en que una clase proporciona servicios a otras clases.

Referencia

Una referencia es como una etiqueta que se encuentra atada a una instancia de una clase. La referencia tiene un nombre particular. C# utiliza una referencia para encontrar el camino a la instancia de la clase y utilizar sus métodos y datos. Una referencia puede ser asignada a otra. Si hace esto, el resultado que obtiene son dos etiquetas que hacen referencia un único objeto en memoria.

Reutilización de código

Un desarrollador debe tomar medidas para asegurarse de que un determinado fragmento del programa se escriba solamente una vez en todo el código del programa. Esto generalmente se logra estableciendo el código dentro de métodos e invocándolos cuando lo necesitemos. El uso de jerarquías de clases también es una forma de reutilizar el código. Sólo tiene que anular los métodos que desea actualizar.

Resaltado de sintáxis

Algunos editores de programación (por ejemplo, Visual Studio) muestran los diferentes elementos de un programa en colores diferentes, para que sea más fácil para el programador seguir y comprender el código. Las palabras reservadas son mostradas en azúl, las cadenas de texto en rojo y los comentarios en verde. Tenga en cuenta que los colores son agregados por el editor, y no hay nada realmente en el archivo de código fuente de C# que determine el color del texto.

Seguridad de tipos - Typesafe

Hemos visto que C # es bastante quisquilloso cuando intenta combinar elementos que no deberían combinarse. Pruebe a intentar establecer un valor punto flotante dentro de una variable de tipo entero, y el compilador le advertirá de que no puede realizar esa acción. La razón de esto es que los desarrolladores del lenguaje han notificado los errores más comunes que se dan en la programación, y han previsto que estos errores se detecten antes de que se ejecute el programa, y no justo después cuando el programa se ha bloqueado y ya no tiene remedio. Uno de estos errores es el de utilizar valores o elementos en contextos donde no es conveniente o no tiene sentido

hacerlo (establecer una cadena de texto en una variable booleana), o en casos en donde se pueda producir una pérdida de datos o precisión (establecer un valor de tipo `double` en una variable de tipo `byte`). Esta clase de prevenciones se deben a la *seguridad de tipos* y C# es muy estricto en este respecto. Algunos otros lenguajes son más flexibles cuando se trata de combinar cosas, y trabajan sobre la base de que el programador sabe lo que hace. Suponen que solo porque el código ha sido escrito para hacer algo, esa cosa debe hacer lo correcto.

C# pone mucha atención en esto (tanto como yo lo estoy haciendo en mi libro). Creo que es importante que los desarrolladores reciban toda la ayuda que puedan para evitar que hagan cosas estúpidas, y un lenguaje que impida combinar elementos, de una manera que podría no ser sensata es algo positivo.

Por supuesto, si realmente quiere imponer su voluntad ante el compilador y obligarlo a compilar su código a pesar de que pueda tener problemas de seguridad de tipos, puede hacerlo mediante el uso de casting.

Sobrecarga - Overload

Un método está sobrecargado cuando otro con el mismo nombre, pero un conjunto diferente de parámetros se declara dentro de la misma clase. Los métodos son sobrecargados cuando existe más de una forma de proporcionar la información para realizar una acción en particular, por ejemplo, una fecha puede ser establecida proporcionando la información del día, mes y año, o mediante una cadena de texto o a través de un simple valor de tipo entero que indique el número de días transcurridos desde el día 1 de enero. Se podrían proporcionar estos tres métodos diferentes sobrecargados para establecer la fecha. En ese caso, se diría que el método `EstablecerFecha` ha sido sobrecargado.

Stream

Un stream es un objeto que representa una conexión a algo que va a transferir datos para nosotros. Esa transferencia de datos podría producirse en un archivo de disco, en un puerto de red o incluso en la consola del sistema. Los streams proporcionan una visión genérica de los repositorios y los orígenes de datos, y evitan que el programador tenga que ocuparse de los detalles específicos del sistema.

Subíndice

Es un valor que se usa para identificar el elemento en una matriz. Este valor debe ser de tipo entero. Los subíndices en C# siempre comienzan en 0 (esto localiza, confusamente, el primer elemento de la matriz) y se extienden hasta el tamaño de la matriz menos 1. Esto significa que, si se crea una

matriz de cuatro elementos, los elementos en la matriz tendrán los valores subíndice de 0,1,2 y 3. La mejor forma de considerar un subíndice es teniendo en cuenta la distancia que tiene que recorrer en sentido inverso en la matriz para obtener el elemento que desea. Lo que significa que el primer elemento de la matriz debe tener un valor subíndice de 0.

This

`this` es una palabra reservada en C# que tiene diferentes significados según el contexto en el que se da. Esta es utilizada en un constructor de una clase para invocar a otro constructor. También se utiliza como referencia a la instancia actual de la clase, para su uso en métodos no estáticos que se ejecutan dentro de esa instancia.

Tipo valor

Un tipo valor almacena un valor simple. Los tipos valor son pasados como valores en las llamadas a métodos y sus valores son copiados en la asignación; es decir, `x = y` hace que el valor en `y` sea copiado en `x`. los cambios que se produzcan en el valor de `x` no afectarán al valor de `y`. Tenga en cuenta que esto contrasta con los tipos referencia, donde el resultado de la asignación anterior haría que `x` e `y` se refieran a la misma instancia.

Indice

(

/

;

{

+

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X