Introducción

A la Programación
Lógica y Diseño

JOYCE FARRELL

7a. Ed.

7a. Ed.

Introducción a la Programación Lógica y Diseño

JOYCE FARRELL

Traductor:
Jorge Alberto Velázquez Arellano

Revisión técnica:
Fabiola Ocampo Botello
José Sánchez Juárez
Roberto De Luna Caballero
Profesores de la Escuela Superior de Cómputo
Instituto Politécnico Nacional

CENGAGE Learning®

Australia • Brasil • Corea • España • Estados Unidos • Japón • México • Reino Unido • Singapur

CENGAGE Learning®

Introducción a la Programación Lógica y Diseño
7a. Ed.
Joyce Farrell

Presidente de Cengage Learning Latinoamérica:
Fernando Valenzuela Migoya

Director editorial, de producción y de plataformas digitales para Latinoamérica:
Ricardo H. Rodríguez

Gerente de procesos para Latinoamérica:
Claudia Islas Licona

Gerente de manufactura para Latinoamérica:
Raúl D. Zendejas Espejel

Gerente editorial de contenidos en español:
Pilar Hernández Santamarina

Gerente de proyectos especiales:
Luciana Rabuffetti

Coordinador de manufactura:
Rafael Pérez González

Editores:
Ivonne Arciniega Torres
Timoteo Eliosa García

Diseño de portada:
Lisa Kuhn/Curio Press, LLC,
www.curiopress.com

Imagen de portada:
© Leigh Prather/Veer

Composición tipográfica:
Rogelio Raymundo Reyna Reynoso

Traducido del libro *Programming Logic and Design, Introductory version*
Seventh edition
Joyce Farrell
Publicado en inglés por Course Technology, una compañía de Cengage Learning © 2013
ISBN: 978-1-133-52651-3

Datos para catalogación bibliográfica:
Farrell, Joyce
Introducción a la Programación Lógica y Diseño,
7a. Ed.
ISBN: 978-607-481-904-5

Visite nuestro sitio en:
http://latinoamerica.cengage.com

Impreso en México
1 2 3 4 5 6 7 15 14 13 12

Contenido

CAPÍTULO 1

Una revisión de las computadoras y la programación

En este capítulo usted aprenderá sobre:

- Sistemas de cómputo
- Lógica de un programa simple
- Los pasos que se siguen en el ciclo de desarrollo del programa
- Declaraciones de seudocódigo y símbolos de diagramas de flujo
- Usar un valor centinela para terminar un programa
- Programación y ambientes de usuario
- La evolución de los modelos de programación

Comprensión de los sistemas de cómputo

Un **sistema de cómputo** es una combinación de todos los componentes que se requieren para procesar y almacenar datos usando una computadora. Todos los sistemas de cómputo están compuestos por múltiples piezas de hardware y software.

- **Hardware** es el equipo o los dispositivos físicos asociados con una computadora. Por ejemplo, todos los teclados, ratones, altavoces e impresoras son hardware. Los dispositivos se manufacturan en forma diferente para las computadoras mainframe grandes, las laptops e incluso para las computadoras más pequeñas que están incorporadas en los productos como automóviles y termostatos, pero los tipos de operaciones que efectúan las computadoras de distintos tamaños son muy parecidos. Cuando se piensa en una computadora con frecuencia son sus componentes físicos los que llegan a la mente, pero para que sea útil necesita más que dispositivos; requiere que se le den instrucciones. Del mismo modo en que un equipo de sonido no hace mucho hasta que se le incorpora la música, el hardware de computadora necesita instrucciones que controlen cómo y cuándo se introducen los datos, cómo se procesan y la forma en que se les da salida o se almacenan.
- **Software** son las instrucciones de la computadora que dicen al hardware qué hacer. El software son **programas** o conjuntos de instrucciones escritos por programadores. Usted puede comprar programas previamente escritos que se han almacenado en un disco o descargarlos de la web. Por ejemplo, en los negocios se utilizan programas de procesamiento de palabras y de contabilidad y los usuarios ocasionales de computadoras disfrutan los que reproducen música y juegos. De manera alternativa, usted puede escribir sus propios programas. Cuando escribe las instrucciones de un software se dice que está **programando.** Este libro se enfoca en el proceso de programación.

El software puede clasificarse en dos extensas categorías:

- **Software de aplicación,** que abarca todos los programas que se aplican para una tarea, como los procesadores de palabras, las hojas de cálculo, los programas de nómina e inventarios, e incluso los juegos.
- **Software de sistema,** que incluye los programas que se usan para manejar una computadora, entre los que se encuentran los sistemas operativos como Windows, Linux o UNIX.

Este libro se enfoca en la lógica que se usa para escribir programas de software de aplicación, aunque muchos de los conceptos se aplican a ambos tipos de software.

Juntos, el hardware y el software llevan a cabo tres operaciones importantes en la mayoría de los programas:

- **Entrada:** los elementos de datos entran en el sistema de cómputo y se colocan en la memoria, donde pueden ser procesados. Los dispositivos de hardware que efectúan operaciones de entrada incluyen teclados y ratones. Los **elementos de datos** constan de todo el texto, los números y otras materias primas que se introducen en una computadora y son procesadas por ella. En los negocios, muchos elementos de datos que se usan son los hechos y las cifras sobre entidades como productos, clientes y personal. Sin embargo, los datos también pueden incluir imágenes, sonidos y movimientos del ratón que el usuario efectúa.
- **Procesamiento:** procesar los elementos de datos implica organizarlos o clasificarlos, comprobar su precisión o realizar cálculos con ellos. El componente de hardware que realiza estas tareas es la **unidad central de procesamiento** o **CPU** (siglas del inglés *central processing unit*).

- **Salida:** después de que los elementos de datos se han procesado, la información resultante por lo general se envía a una impresora, un monitor o algún otro dispositivo de salida de modo que las personas vean, interpreten y usen los resultados. Los profesionales de la programación con frecuencia emplean el término *datos* para los elementos de entrada y el de **información** para los datos que se han procesado y que han salido. En ocasiones usted coloca estos datos de salida en **dispositivos de almacenamiento,** como discos o medios flash (*flash media*). Las personas no pueden leer los datos en forma directa desde tales dispositivos, pero éstos contienen información que puede recuperarse posteriormente. Cuando se envía una salida a un dispositivo de almacenamiento en ocasiones se usa después como entrada para otro programa.

Las instrucciones para el ordenador se escriben en un **lenguaje de programación** como Visual Basic, C#, C++ o Java. Del mismo modo en que algunas personas hablan inglés y otras japonés, los programadores escriben en diferentes lenguajes. Algunos de ellos trabajan de manera exclusiva en uno de ellos mientras que otros conocen varios y usan aquel que sea más adecuado para la tarea que se les presenta.

Las instrucciones que usted escribe usando un lenguaje de programación constituyen un **código de programa;** cuando las escribe está **codificando el programa.**

Cada lenguaje de programación tiene reglas que rigen el uso de las palabras y la puntuación. Estas reglas se llaman **sintaxis** del lenguaje. Los errores en el uso de un lenguaje son **errores de sintaxis.** Si usted pregunta: "¿Cómo otengo forma la guardarlo de?" en español, casi todas las personas pueden imaginar lo que probablemente usted quiso decir, aun cuando no haya usado una sintaxis apropiada en español: ha mezclado el orden de las palabras y ha escrito mal una de ellas. Sin embargo, las computadoras no son tan inteligentes como la mayoría de las personas; en este caso, bien podría haber preguntado a la computadora: "¿Xpu mxv ort dod nmcad bf B?". A menos que la sintaxis sea perfecta, la computadora no puede interpretar en absoluto la instrucción del lenguaje de programación.

Cuando usted escribe un programa por lo general transmite sus instrucciones usando un teclado. Cuando lo hace, las instrucciones se almacenan en la **memoria de la computadora,** que es un almacenamiento interno temporal. La **memoria de acceso aleatorio** o **RAM** es una forma de memoria interna volátil. Los programas que "corren" (es decir, se ejecutan) en la actualidad y los elementos de datos que se usan se almacenan en la RAM para que sea posible tener un acceso rápido a ellos. El almacenamiento interno es **volátil,** es decir, su contenido se pierde cuando la computadora se apaga o se interrumpe la energía. Por lo general, usted desea recuperar y quizá modificar más tarde las instrucciones almacenadas, de modo que también tiene que guardarlas en un dispositivo de almacenamiento permanente, como un disco. Estos dispositivos se consideran **no volátiles,** esto es, su contenido es permanente y se conserva aun cuando la energía se interrumpa. Si usted ha experimentado una interrupción de la energía mientras trabajaba en una computadora pero pudo recuperar su información cuando aquélla se restableció, no se debió a que su trabajo todavía se encontrara en la RAM. Su sistema se ha configurado para guardar automáticamente su trabajo a intervalos regulares en un dispositivo de almacenamiento no volátil.

Después de que se ha escrito un programa usando declaraciones de un lenguaje de programación y se ha almacenado en la memoria, debe traducirse a un **lenguaje de máquina** que representa los millones de circuitos encendidos/apagados dentro de la computadora. Sus declaraciones en lenguaje de programación se llaman **código fuente** y las traducidas al lenguaje de máquina se denominan **código objeto.**

Cada lenguaje de programación usa una pieza de software llamada **compilador** o **intérprete** para traducir su código fuente al lenguaje de máquina. Este último también se llama **lenguaje binario** y se representa como una serie de 0 (ceros) y 1 (unos). El compilador o intérprete que traduce el código indica si cualquier componente del lenguaje de programación se ha usado de manera incorrecta. Los errores de sintaxis son relativamente fáciles de localizar y corregir debido a que el compilador o intérprete los resalta. Si usted escribe un programa de computadora usando un lenguaje como C++ pero deletrea en forma incorrecta una de sus palabras o invierte el orden apropiado de dos de ellas, el software le hace saber que encontró un error desplegando un mensaje tan pronto como usted intenta traducir el programa.

Aunque hay diferencias en la forma en que trabajan los compiladores y los intérpretes, su función básica es la misma: traducir sus declaraciones de programación en un código que la computadora pueda usar. Cuando usted usa un compilador, un programa entero se traduce antes de que pueda ejecutarse; cuando utiliza un intérprete, cada instrucción es traducida justo antes de la ejecución. Por lo general usted no elige cuál tipo de traducción usar, esto depende del lenguaje de programación. Sin embargo, hay algunos lenguajes que disponen tanto de compiladores como de intérpretes.

Después de que el código fuente es traducido con éxito al lenguaje de máquina, la computadora puede llevar a cabo las instrucciones del programa. Un programa **corre** o se **ejecuta** cuando se realizan las instrucciones**.** En un programa típico se aceptará alguna entrada, ocurrirá algún procesamiento y se producirá alguna salida.

Además de los lenguajes de programación exhaustivos populares como Java y C++, muchos programadores usan **lenguajes de programación interpretados** (que también se llaman **lenguajes de programación de scripting** o **lenguajes de script**) como Python, Lua, Perl y PHP. Los scripts escritos en estos lenguajes por lo general pueden mecanografiarse en forma directa desde un teclado y almacenarse como texto en lugar de como archivos ejecutables binarios. Los lenguajes de script son interpretados línea por línea cada vez que se ejecuta el programa, en lugar de ser almacenados en forma compilada (binaria). Aun

DOS VERDADES Y UNA MENTIRA

Comprensión de los sistemas de cómputo

En cada sección "Dos verdades y una mentira", dos de las afirmaciones numeradas son verdaderas y una es falsa. Identifique la que es falsa y explique por qué lo es.

1. El hardware es el equipo o los dispositivos asociados con una computadora. El software son las instrucciones.
2. Las reglas gramaticales de un lenguaje de programación de computadoras constituyen su sintaxis.
3. Usted escribe programas usando el lenguaje de máquina y el software de traducción convierte las declaraciones a un lenguaje de programación.

La afirmación falsa es la número 3. Se escriben programas usando un lenguaje de programación como Visual Basic o Java, y un programa de traducción (llamado compilador o intérprete) convierte las declaraciones en lenguaje de máquina, el cual se compone de 0 (ceros) y 1 (unos).

así, con todos los lenguajes de programación cada instrucción debe traducirse al lenguaje de máquina antes de que pueda ejecutarse.

Comprensión de la lógica de programa simple

Un programa con errores de sintaxis no puede traducirse por completo ni ejecutarse. Uno que no los tenga es traducible y puede ejecutarse, pero todavía podría contener **errores lógicos** y dar como resultado una salida incorrecta. Para que un programa funcione en forma apropiada usted debe desarrollar una **lógica** correcta, es decir, escribir las instrucciones en una secuencia específica, no dejar fuera ninguna instrucción y no agregar instrucciones ajenas.

Suponga que indica a alguien que haga un pastel de la siguiente manera:

```
Consiga un tazón
Revuelva
Agregue dos huevos
Agregue un litro de gasolina
Hornee a 350º por 45 minutos
Agregue tres tazas de harina
```

No lo haga
¡No hornee un pastel como éste!

Las instrucciones peligrosas para hornear un pastel se muestran con un icono “No lo haga”. Verá este icono cuando se presente en el libro una práctica de programación no recomendada que se usa como ejemplo de lo que *no* debe hacerse.

Aun cuando la sintaxis de las instrucciones para hornear un pastel es correcta en español, están fuera de secuencia; faltan algunas y otras pertenecen a procedimientos distintos que no tienen nada que ver con hornear un pastel. Si las sigue no hará un pastel que sea posible ingerir y quizá termine por ser un desastre. Muchos errores lógicos son más difíciles de localizar que los de sintaxis; es más fácil determinar si la palabra “huevos” en una receta está escrita en forma incorrecta que decir si hay demasiados o si se agregaron demasiado pronto.

Del mismo modo en que las instrucciones para hornear pueden proporcionarse en chino mandarín, urdu o inglés, la lógica de programa puede expresarse en forma correcta en cualquier cantidad de lenguajes de programación. Debido a que este libro no se enfoca en algún lenguaje específico, los ejemplos de programación pudieron escribirse en Visual Basic, C++ o Java. Por conveniencia, ¡en este libro las instrucciones se han escrito en español!

Después de aprender francés, usted automáticamente conoce, o puede imaginar con facilidad, muchas palabras en español. Del mismo modo, después de aprender un lenguaje de programación, es mucho más fácil entender otros lenguajes.

Los programas de computadora más sencillos incluyen pasos que ejecutan la entrada, el procesamiento y la salida. Suponga que desea escribir un programa para duplicar cualquier número que proporcione. Puede escribirlo en un lenguaje como Visual Basic o Java, pero si lo escribiera con declaraciones en inglés, se verían así:

```
input myNumber
set myAnswer = myNumber * 2
output myAnswer
```

El proceso de duplicar el número incluye tres instrucciones:

- La instrucción `input myNumber` es un ejemplo de una operación de entrada. Cuando la computadora interpreta esta instrucción sabe que debe buscar un dispositivo de entrada para obtener un número. Cuando usted trabaja en un lenguaje de programación específico, escribe instrucciones que indican a la computadora a cuál dispositivo se tiene acceso para obtener la entrada. Por ejemplo, cuando un usuario introduce un número como los datos para un programa podría hacer clic en el número con un ratón, mecanografiarlo en un teclado o hablar en un micrófono. Sin embargo, es lógico que no importa cuál dispositivo de hardware se use siempre que la computadora sepa que debe aceptar un número. Cuando el número se recupera desde un dispositivo de entrada, se coloca en la memoria de la computadora en una variable llamada `myNumber`. Una **variable** es una ubicación de memoria nombrada cuyo valor puede variar; por ejemplo, el valor de `myNumber` podría ser 3 cuando el programa se usa por primera vez y 45 cuando se usa la siguiente vez. En este libro, los nombres de las variables no llevarán espacios; por ejemplo, se usará `myNumber` en lugar de `my Number`.

Desde una perspectiva lógica, cuando usted introduce, procesa o da salida a un valor, el dispositivo de hardware es irrelevante. Lo mismo sucede en su vida diaria. Si sigue la instrucción "Obtener huevos para el pastel", en realidad no importa si los compra en una tienda o los recolecta de sus propias gallinas; usted los consigue de cualquier forma. Podría haber diferentes consideraciones prácticas para obtenerlos, del mismo modo que las hay para obtener los datos de una base de datos grande en contraposición a obtenerlos de un usuario inexperto que trabaja en casa en una laptop. Por ahora, este libro sólo se interesa en la lógica de las operaciones, no en detalles menores.

- La instrucción `set myAnswer = myNumber * 2` es un ejemplo de una operación de procesamiento. En la mayoría de los lenguajes de programación se usa un asterisco para indicar una multiplicación, de modo que esta instrucción significa "Cambiar el valor de la ubicación de memoria `myAnswer` para igualar el valor en la ubicación de memoria `myNumber` por dos". Las operaciones matemáticas no son el único tipo de operaciones de procesamiento, pero son típicas. Como sucede con las operaciones de entrada, el tipo de hardware que se usa para el procesamiento es irrelevante; después de que usted escribe un programa, éste puede usarse en computadoras de diferentes marcas, tamaños y velocidades.

- En el programa para duplicar un número, la instrucción `output myAnswer` es un ejemplo de una operación de salida. Dentro de un programa particular, esta declaración podría causar que la salida aparezca en el monitor (digamos, una pantalla plana de plasma o una de tubo de rayos catódicos), que vaya a una impresora (láser o de inyección de tinta) o que se escriba en un disco o un DVD. La lógica del proceso de salida es la misma sin importar qué dispositivo de hardware se use. Cuando se ejecuta esta instrucción, el valor almacenado en la memoria en la ubicación llamada `myAnswer` se envía a un dispositivo de salida. (El valor de salida también permanece en la memoria hasta que se almacena algo más en la misma ubicación o se interrumpe la energía eléctrica.)

La memoria de la computadora consiste en millones de ubicaciones numeradas donde es posible almacenar datos. La ubicación de memoria de `myNumber` tiene una ubicación numérica específica, pero cuando usted escribe programas rara vez necesita preocuparse por el valor de la dirección de memoria; en cambio, usa el nombre fácil de recordar que creó. Los programadores de computadoras con frecuencia se refieren a las direcciones de memoria usando la notación hexadecimal, o en base 16. Con este sistema podrían utilizar un valor como 42FF01A para referirse a una dirección de memoria. A pesar del uso de letras, dicha dirección todavía es un número hexadecimal. El apéndice A contiene información sobre este sistema de numeración.

DOS VERDADES Y UNA MENTIRA

Comprensión de la lógica de programa simple

1. Un programa con errores de sintaxis puede ejecutarse pero podría generar resultados incorrectos.
2. Aunque la sintaxis de los lenguajes de programación difiere, la misma lógica de programa puede expresarse en diferentes lenguajes.
3. Los programas de computadora más sencillos incluyen pasos que efectúan entrada, procesamiento y salida.

La afirmación falsa es la número 1. Un programa con errores de sintaxis no puede ejecutarse; uno que no tenga este tipo de errores puede ejecutarse, pero produciría resultados incorrectos.

Comprensión del ciclo de desarrollo del programa

El trabajo de un programador implica escribir instrucciones (como las del programa para duplicar números en la sección anterior), pero por lo general un profesional no sólo se sienta ante un teclado de computadora y comienza a mecanografiar. La figura 1-1 ilustra el **ciclo de desarrollo del programa,** que se divide al menos en siete pasos:

1. Entender el problema.
2. Planear la lógica.
3. Codificar el programa.
4. Usar software (un compilador o intérprete) para traducir el programa a lenguaje de máquina.
5. Probar el programa.
6. Poner el programa en producción.
7. Mantener el programa.

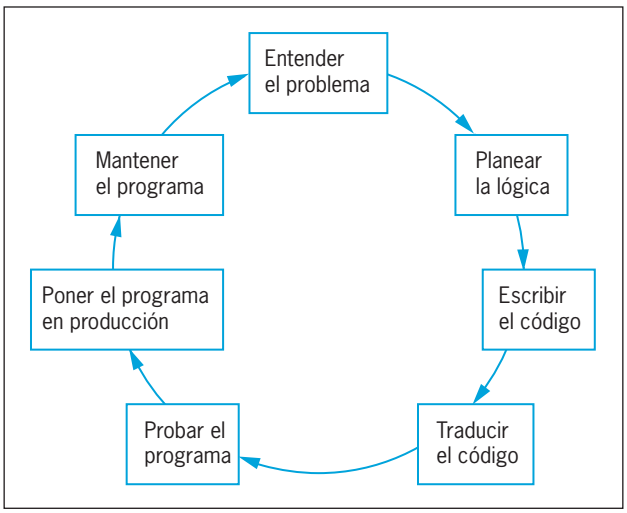

Figura 1-1 El ciclo de desarrollo del programa

Entender el problema

Los programadores profesionales escriben programas para satisfacer las necesidades de otras personas, llamadas **usuarios** o **usuarios finales.** Entre los ejemplos de usuarios finales estaría un departamento de recursos humanos que necesita una lista impresa de todos los empleados, un área de facturación que desea un listado de los clientes que se han retrasado en sus pagos 30 días o más, o un departamento de pedidos que requiere un sitio web para proporcionar a los compradores un carrito de compras en línea. Debido a que los programadores brindan un servicio a estos usuarios deben comprender primero lo que éstos desean. Cuando usted corre un programa con frecuencia piensa en la lógica como un ciclo de operaciones de entrada-procesamiento-salida; pero cuando planea un programa piensa primero en la salida. Después de entender cuál es el resultado deseado puede planear los pasos de entrada y procesamiento para lograrlo.

Suponga que el director de recursos humanos dice a un programador: "Nuestro departamento necesita una lista de todos los empleados que han estado aquí por más de cinco años, porque queremos invitarlos a una cena especial de agradecimiento". En apariencia esta es una solicitud sencilla. Sin embargo, un programador experimentado sabrá que la solicitud está incompleta. Por ejemplo, quizá no sepa las respuestas a las siguientes preguntas sobre cuáles empleados incluir:

- ¿El director desea una lista sólo de empleados de tiempo completo o de tiempo completo y de medio tiempo juntos?
- ¿Desea incluir personas que han trabajado para la compañía con una base contractual mensual durante los pasados cinco años o sólo los empleados permanentes regulares?
- ¿Los empleados necesitan haber trabajado para la organización por cinco años hasta el día de hoy, hasta la fecha de la cena o en alguna otra fecha límite?
- ¿Qué pasa con un empleado que trabajó tres años, tomó una licencia de dos años y volvió a trabajar por tres años?

El programador no puede tomar ninguna de estas decisiones; el usuario (en este caso, el director de recursos humanos) debe abordar estas preguntas.

Tal vez aún se requiera tomar otras decisiones, por ejemplo:

- ¿Qué datos deben incluirse para cada empleado en la lista? ¿Es preciso anotar el nombre y los apellidos? ¿Los números de seguro social? ¿Números telefónicos? ¿Direcciones?
- ¿La lista debe estar en orden alfabético? ¿Por número de identificación del empleado? ¿En orden de años de servicio? ¿Algún otro orden?
- ¿Los empleados deberían agruparse con algún criterio, como número de departamento o años de servicio?

A menudo se proporcionan algunas piezas de documentación para ayudar al programador a entender el problema. La **documentación** consiste en todo el papeleo de soporte para un programa; podría incluir elementos como las solicitudes originales para el programa de los usuarios, muestras de salida y descripciones de los elementos de datos disponibles para la entrada.

Entender por completo el problema es uno de los aspectos más difíciles de la programación. En cualquier trabajo, la descripción de lo que el usuario necesita puede ser imprecisa; peor aún, los usuarios quizá no sepan qué desean en realidad, y los que piensan que saben a menudo cambian de opinión después de ver una muestra de salida. ¡Un buen programador es en parte consejero y en parte detective!

Planear la lógica

El corazón del proceso de programación se encuentra en la planeación de la lógica del programa. Durante esta fase, el programador planifica los pasos del mismo, decidiendo cuáles incluir y cómo ordenarlos. Usted puede visualizar la solución de un problema de muchas maneras. Las dos herramientas de programación más comunes son los diagramas de flujo y el seudocódigo; ambas implican escribir los pasos del programa en inglés, del mismo modo en que planearía un viaje en papel antes de subirse al automóvil o el tema de una fiesta antes de comprar alimentos y recuerdos.

Quizá usted haya escuchado a los programadores referirse a la planeación de un programa como "desarrollar un algoritmo". Un **algoritmo** es la secuencia de pasos necesarios para resolver cualquier problema.

Además de los diagramas de flujo y el seudocódigo, los programadores usan una variedad de herramientas distintas para desarrollar el programa. Una de ellas es la **gráfica IPO,** que define las tareas de entrada, procesamiento y salida. Algunos programadores orientados hacia los objetos también usan **gráficas TOE,** que listan tareas, objetos y eventos.

El programador no debe preocuparse por la sintaxis de algún lenguaje en particular durante la etapa de planeación, sino enfocarse en averiguar qué secuencia de eventos llevará desde la entrada disponible hasta la salida deseada. La planeación de la lógica incluye pensar con

cuidado en todos los valores de datos posibles que un programa podría encontrar y cómo desea que éste maneje cada escenario. El proceso de recorrer en papel la lógica de un programa antes de escribirlo en realidad se llama **prueba de escritorio (*desk-checking*).** Aprenderá más sobre la planeación de la lógica a lo largo de este libro; de hecho, éste se enfoca casi de manera exclusiva en este paso crucial.

Codificación del programa

Sólo después de que se ha desarrollado la lógica el programador puede escribir el código fuente. Hay cientos de lenguajes de programación disponibles. Los programadores eligen lenguajes particulares debido a que algunos incorporan capacidades que los hacen más eficientes que otros para manejar ciertos tipos de operaciones. A pesar de sus diferencias, los lenguajes de programación son bastante parecidos en sus capacidades básicas; cada uno puede manejar operaciones de entrada, procesamiento aritmético, operaciones de salida y otras funciones estándares. La lógica que se desarrolla para resolver un problema de programación puede ejecutarse usando cualquier cantidad de lenguajes. Sólo después de elegir alguno el programador debe preocuparse por la puntuación y la ortografía correctas de los comandos; en otras palabras, por usar la *sintaxis* correcta.

Algunos programadores experimentados combinan con éxito en un paso la planeación de la lógica y la codificación del programa. Esto funciona para planear y escribir un programa muy sencillo, del mismo modo en que usted puede planear y escribir una postal para un amigo en un solo paso. Sin embargo, la redacción de un buen ensayo semestral o un guión cinematográfico requiere planeación y lo mismo sucede con la mayor parte de los programas.

¿Cuál paso es más difícil: planear la lógica o codificar el programa? Ahora mismo quizá le parezca que escribir en un lenguaje de programación es una tarea muy difícil, considerando todas las reglas de ortografía y sintaxis que debe aprender. Sin embargo, en realidad el paso de planeación es más difícil. ¿Qué es más complicado: pensar en los giros de la trama de una novela de misterio que es un éxito de ventas o escribir la traducción del inglés al español de una novela que ya se ha escrito? ¿Y quién cree que recibe más paga, el escritor que crea la trama o el traductor? (¡Haga la prueba pidiendo a algunos amigos que nombren a algún traductor famoso!)

Uso de software para traducir el programa al lenguaje de máquina

Aun cuando hay muchos lenguajes de programación, cada computadora conoce sólo uno: su lenguaje de máquina, que consiste en 1 (unos) y 0 (ceros). Las computadoras entienden el lenguaje de máquina porque están formadas por miles de diminutos interruptores eléctricos, cada uno de los cuales presenta un estado de encendido o apagado, que se representa con 1 o 0, respectivamente.

Lenguajes como Java o Visual Basic están disponibles para los programadores debido a que alguien ha escrito un programa traductor (un compilador o intérprete) que cambia el **lenguaje de programación de alto nivel** en inglés del programador en un **lenguaje de máquina de bajo nivel** que la computadora entiende. Cuando usted aprende la sintaxis de un lenguaje de programación, los comandos funcionan en cualquier máquina en la que el software del lenguaje se haya instalado. Sin embargo, sus comandos son traducidos entonces al lenguaje de máquina, que es distinto en las distintas marcas y modelos de computadoras.

Si usted escribe en forma incorrecta una declaración de programación (por ejemplo, escribe mal una palabra, usa alguna que no existe o utiliza gramática "ilegal"), el programa traductor no sabe cómo proceder y emite un mensaje al detectar un error de sintaxis. Aunque nunca es deseable cometerlos, los errores de sintaxis no son una preocupación importante para los programadores porque el compilador o intérprete los detecta y muestra un mensaje que les notifica el problema. La computadora no ejecutará un programa que contenga aunque sea sólo un error de sintaxis.

Por lo común, un programador desarrolla la lógica, escribe el código y compila el programa, recibiendo una lista de errores de sintaxis. Entonces los corrige y compila el programa de nuevo. La corrección del primer conjunto de errores con frecuencia revela otros nuevos que al principio no eran evidentes para el compilador. Por ejemplo, si usted usa un compilador en español y envía la declaración *El prro persiguen al gato*, el compilador al principio señalaría sólo un error de sintaxis. La segunda palabra, *prro*, es ilegal porque no forma parte del español. Sólo después de corregirla a *perro* el compilador hallaría otro error de sintaxis en la tercera palabra, *persiguen*, porque es una forma verbal incorrecta para el sujeto *perro*. Esto no significa que *persiguen* necesariamente sea la palabra equivocada. Quizá *perro* es incorrecto; tal vez el sujeto debería ser *perros*, en cuyo caso *persiguen* sería correcto. Los compiladores no siempre saben con exactitud qué quiere usted ni cuál debería ser la corrección apropiada, pero sí saben cuando algo anda mal con su sintaxis.

Cuando un programador escribe un programa tal vez necesite recompilar el código varias veces. Un programa ejecutable sólo se crea cuando el código no tiene errores de sintaxis. Después de que un programa se ha traducido al lenguaje de máquina, se guarda y puede ser ejecutado cualquier número de veces sin repetir el paso de traducción. Usted sólo necesita retraducir su código si hace cambios en las declaraciones de su código fuente. La figura 1-2 muestra un diagrama de este proceso en su totalidad.

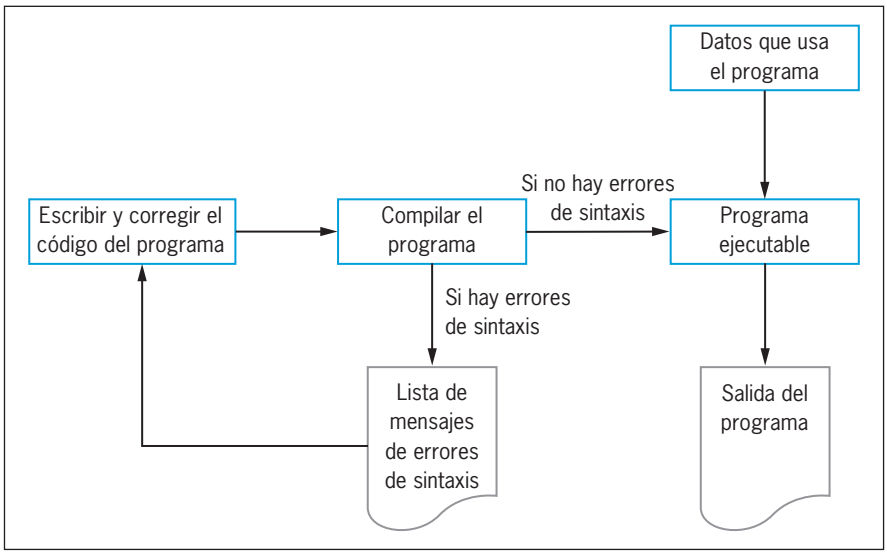

Figura 1-2 Creación de un programa ejecutable

Prueba del programa

Un programa que no tiene errores de sintaxis no necesariamente está libre de errores lógicos. Un error lógico resulta cuando se utiliza una declaración correcta desde el punto de vista sintáctico pero equivocada para el contexto actual. Por ejemplo, la declaración en español *El perro persigue al gato*, aunque sintácticamente es perfecta, no es correcta desde una perspectiva lógica si el perro persigue una pelota o si el gato es el agresor.

Una vez que un programa queda limpio de errores de sintaxis el programador puede probarlo, es decir, ejecutarlo con algunos datos de muestra para ver si los resultados son lógicamente correctos. Recuerde el programa para duplicar un número:

```
input myNumber
set myAnswer = myNumber * 2
output myAnswer
```

Si usted ejecuta el programa, proporciona el valor 2 como entrada para el mismo y se despliega la respuesta 4, ha ejecutado una corrida de prueba exitosa del programa.

Sin embargo, si se despliega la respuesta 40, quizá el programa contenga un error lógico. Tal vez usted tecleó mal la segunda línea del código con un cero extra, de modo que el programa se lee:

```
input myNumber
set myAnswer = myNumber * 20
output myAnswer
```

No lo haga
El programador tecleó 20 en lugar de 2.

Escribir 20 en lugar de 2 en la declaración de multiplicación causó un error lógico. Observe que desde el punto de vista sintáctico no hay nada incorrecto en este segundo programa (es

igual de razonable multiplicar un número por 20 que por 2) pero si el programador sólo pretende duplicar `myNumber`, entonces ha ocurrido un error lógico.

El proceso de hallar y corregir los errores del programa se llama **depuración.** Usted depura un programa al probarlo usando muchos conjuntos de datos. Por ejemplo, si escribe el programa para duplicar un número, luego introduce 2 y obtiene un valor de salida de 4, esto no necesariamente significa que el programa es correcto. Quizá tecleó por error este programa:


```
input myNumber
set myAnswer = myNumber + 2
output myAnswer
```

No lo haga
El programador tecleó "+" en lugar de "*".

Una entrada de 2 da como resultado una respuesta de 4, pero esto no significa que su programa duplique los números; en realidad sólo les suma 2. Si prueba su programa con datos adicionales y obtiene la respuesta errónea; por ejemplo, si introduce 7 y obtiene una respuesta de 9, sabe que hay un problema con su código.

La selección de los datos de prueba es casi un arte en sí misma y debe hacerse con cuidado. Si el departamento de recursos humanos desea una lista de los nombres de los empleados con antigüedad de cinco años, sería un error probar el programa con un pequeño archivo de muestra que sólo contiene empleados de tiempo indeterminado. Si los empleados más recientes no son parte de los datos que se usan para probar, en realidad no sabe si el programa los habría eliminado de la lista de cinco años. Muchas compañías no saben que su software tiene un problema hasta que ocurre una circunstancia extraña; por ejemplo, la primera vez que un empleado registra más de nueve dependientes, la primera vez que un cliente ordena más de 999 artículos al mismo tiempo o cuando a la internet se le agotan las direcciones IP asignadas, un problema que se conoce como *agotamiento IPV4*.

Poner el programa en producción

Una vez que se ha probado y depurado el programa en forma minuciosa, está listo para que la organización lo use. "Ponerlo en producción" significaría simplemente ejecutarlo una vez, si fue escrito para satisfacer una solicitud del usuario para una lista especial. Sin embargo, el proceso podría llevar meses si el programa se ejecutará en forma regular o si es uno de un gran sistema de programas que se están desarrollando. Quizá las personas que introducirán los datos deben recibir capacitación con el fin de preparar las entradas para el nuevo programa, los usuarios deben recibir instrucción para entender la salida o sea preciso cambiar los datos existentes en la compañía a un formato por completo nuevo para que tengan cabida en dicho programa. Completar la **conversión,** el conjunto entero de acciones que debe efectuar una organización para cambiar al uso de un programa o un conjunto de programas nuevos, en ocasiones puede llevar meses o años.

Mantenimiento del programa

Después de que los programas se colocan en producción, la realización de los cambios necesarios se denomina **mantenimiento.** Puede requerirse por diversas razones: por ejemplo, debido a que se han legislado nuevas tasas de impuestos, se alteró el formato de un archivo de entrada

o el usuario final requiere información adicional no incluida en las especificaciones de salida originales. Con frecuencia, la primera labor de programación que usted lleve a cabo requerirá dar mantenimiento a los programas escritos de manera previa. Cuando dé mantenimiento a los programas que otras personas han escrito apreciará el esfuerzo que hicieron para obtener un código claro, usar nombres de variables razonables y documentar su trabajo. Cuando hace cambios a los programas existentes repite el ciclo de desarrollo. Es decir, debe entender los cambios, luego planearlos, codificarlos, traducirlos y probarlos antes de ponerlos en producción. Si el programa original requiere una cantidad considerable de modificaciones podría ser retirado y empezaría el ciclo de desarrollo del programa para uno nuevo.

DOS VERDADES Y UNA MENTIRA

Comprensión del ciclo de desarrollo del programa

1. Entender el problema que debe resolverse puede ser uno de los aspectos más difíciles de la programación.
2. Las dos herramientas más comunes que se usan en la planeación de la lógica son los diagramas de flujo y el seudocódigo.
3. La elaboración del diagrama de flujo de un programa es un proceso muy diferente si se usa un lenguaje de programación antiguo en lugar de uno más reciente.

La afirmación falsa es la número 3. A pesar de sus diferencias, los lenguajes de programación son bastante parecidos en sus capacidades básicas; cada uno puede manejar operaciones de entrada, procesamiento aritmético, operaciones de salida y otras funciones estándar. La lógica que se ha desarrollado para resolver un problema de programación puede ejecutarse usando cualquier cantidad de lenguajes.

Uso de declaraciones en seudocódigo y símbolos de diagrama de flujo

Cuando los programadores planean la lógica para dar solución a un problema de programación con frecuencia usan dos herramientas: seudocódigo o diagramas de flujo.

- El **seudocódigo** es una representación parecida al inglés de los pasos lógicos que se requieren para resolver un problema. *Seudo* es un prefijo que significa *falso*, y *codificar* un programa significa ponerlo en un lenguaje de programación; por consiguiente, *seudocódigo* simplemente significa *código falso*, o declaraciones que en apariencia se han escrito en un lenguaje de programación pero no necesariamente siguen todas las reglas de sintaxis de alguno en específico.
- Un **diagrama de flujo** es una representación gráfica de lo mismo.

CAPÍTULO 2

Elementos de los programas de alta calidad

En este capítulo usted aprenderá sobre:

- La declaración y el uso de variables y constantes
- La realización de operaciones aritméticas
- Las ventajas de la modularización
- Modularizar un programa
- Las gráficas de jerarquía
- Las características de un buen diseño de programa

La declaración y el uso de variables y constantes

Como aprendió en el capítulo 1, los elementos de datos incluyen todo el texto, los números y otra información que son procesados por una computadora. Cuando usted introduce los elementos de datos en un ordenador, éstos se almacenan en variables en la memoria, donde se procesan y se convierten en información de salida.

Cuando usted escribe programas trabaja con los datos en tres formas diferentes: literales (o constantes no nombradas), variables y constantes nombradas.

Comprensión de las constantes literales y sus tipos de datos

Todos los lenguajes de programación soportan dos amplios tipos de datos: el **numérico** describe los datos que consisten en números y la **cadena** los que no son numéricos. La mayoría de los lenguajes de programación soportan varios tipos de datos adicionales, incluidos los tipos múltiples para valores numéricos de diferentes tamaños y con y sin lugares decimales. Los lenguajes como C++, C#, Visual Basic y Java distinguen entre las variables numéricas **enteras** (número entero) y las variables numéricas de **punto flotante** (fraccionarias) que tienen un punto decimal. (Los números de punto flotante también se conocen como **números reales.**) Por tanto, en algunos lenguajes los valores 4 y 4.3 se almacenarían en tipos diferentes de variables numéricas. Además, muchos lenguajes permiten la distinción entre los valores más pequeños y más grandes que ocupan diferentes cantidades de bytes en la memoria. Usted aprenderá más sobre estos tipos de datos especializados cuando estudie un lenguaje de programación, pero este libro usa los dos más amplios: numérico y cadena.

Cuando usted usa un valor numérico específico, como 43, en un programa, lo escribe usando los dígitos y sin comillas. Un valor numérico específico con frecuencia se llama **constante numérica** (o **constante numérica literal**) debido a que no cambia; un 43 siempre tiene el valor 43. Cuando se almacena un valor numérico en la memoria de la computadora no se introducen o almacenan caracteres adicionales como signos de pesos y comas. Esos caracteres pueden agregarse a la salida con fines de legibilidad, pero no son parte del número.

Un valor de texto específico, o cadena de caracteres, como “Amanda”, es una **constante de cadena** (o **constante de cadena literal**). Las constantes de cadena, a diferencia de las constantes numéricas, aparecen entre comillas en los programas. Los valores de cadena también se llaman **valores alfanuméricos** porque pueden contener caracteres alfabéticos al igual que números y otros caracteres. Por ejemplo, “$3,215.99 U.S.”, incluido el signo de dólares, la coma, el punto, las letras y los números, es una cadena. Aunque las cadenas pueden contener números, los valores numéricos no contienen caracteres alfabéticos. La constante numérica 43 y la constante de cadena “Amanda” son ejemplos de **constantes literales,** no tienen identificadores como las variables.

Trabajo con variables

Las variables son ubicaciones de memoria nombradas cuyo contenido varía o difiere con el tiempo. Por ejemplo, en el programa para duplicar números en la figura 2-1, `myNumber` y `myAnswer` son variables. En cualquier momento en el tiempo, una variable sólo contiene un valor. En ocasiones, `myNumber` contiene 2 y `myAnswer` contiene 4; otras veces, `myNumber` contiene 6 y `myAnswer` 12. La capacidad de las variables de memoria para cambiar de valor es lo que hace que las computadoras y la programación sean valiosas. Debido a que una ubicación de memoria puede usarse en repetidas ocasiones con diferentes valores, es posible escribir las instrucciones del programa una vez y luego usarlas para miles de cálculos separados. *Un* conjunto de instrucciones de nómina en su empresa genera el cheque de pago de cada empleado y *un* conjunto de instrucciones en su compañía eléctrica resulta en la factura de cada hogar.

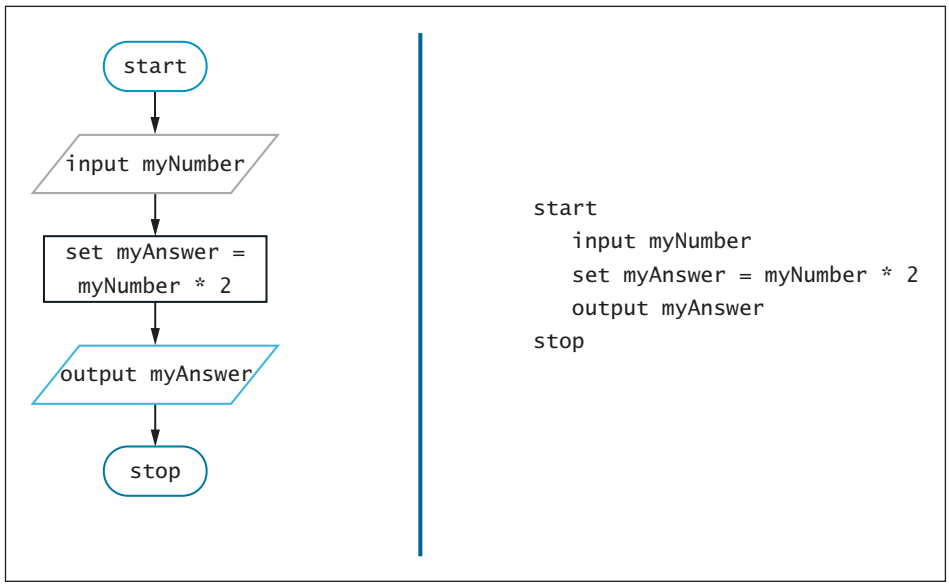

Figura 2-1 Diagrama de flujo y seudocódigo para el programa para duplicar números

En la mayoría de los lenguajes de programación, antes de que usted pueda usar alguna variable debe incluir una declaración para ello. Una **declaración** es un enunciado que proporciona el tipo de datos y un identificador para una variable. Un **identificador** es el nombre de un componente del programa. El **tipo de datos** de un elemento de datos es una clasificación que describe lo siguiente:

- Qué valores puede contener el elemento
- Cómo se almacena el elemento en la memoria de la computadora
- Qué operaciones pueden ejecutarse en el elemento de datos

Como se mencionó antes, casi todos los lenguajes de programación soportan varios tipos de datos, pero en este libro sólo se usarán dos tipos: `num` y `string`.

Cuando usted declara una variable proporciona tanto un tipo de datos como un identificador. De manera opcional, puede declarar un valor inicial para cualquier variable y hacer esto es **inicializar la variable.** Por ejemplo, cada una de las siguientes declaraciones es válida. Dos de ellas incluyen inicializaciones y dos no:

```
num mySalary
num yourSalary = 14.55
string myName
string yourName = "Juanita"
```

La figura 2-2 muestra el programa para duplicar números de la figura 2-1 con las declaraciones agregadas sombreadas. Las variables deben declararse antes de que se usen por primera vez en un programa. Algunos lenguajes requieren que todas se declaren al principio del programa; otros permiten que esto se haga en cualquier parte con tal de que sea antes de su primer uso. Este libro seguirá la convención de declarar todas las variables juntas.

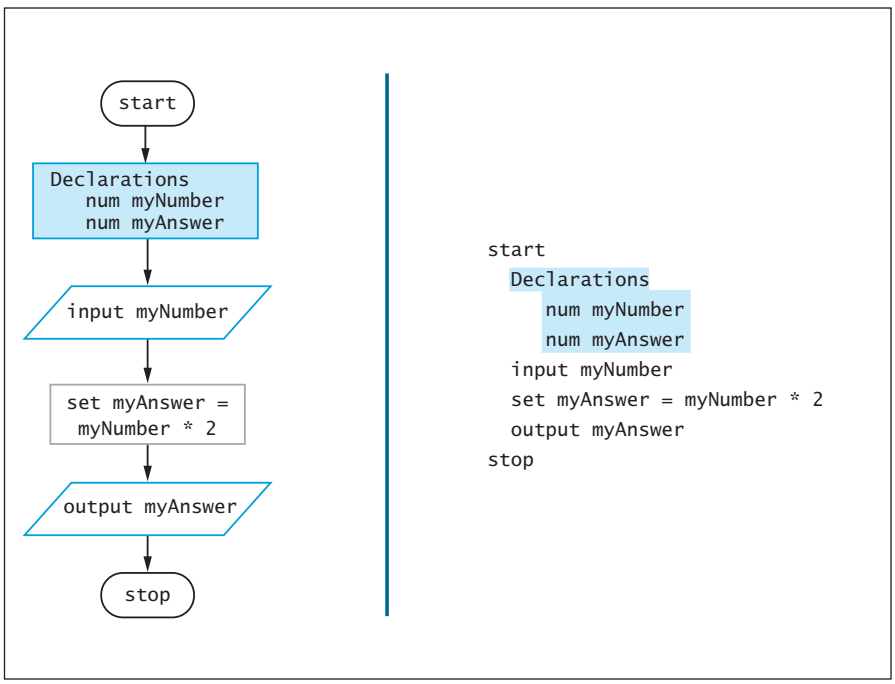

Figura 2-2 Diagrama de flujo y seudocódigo del programa para duplicar números con declaraciones de variables

En muchos lenguajes de programación, si se declara una variable y no se inicializa, ésta contendrá un valor desconocido hasta que se le asigne uno. El valor desconocido de una variable por lo común se llama **basura.** Aunque algunos lenguajes usan un valor por defecto para algunas variables (como asignar 0 a cualquier variable numérica no asignada), en este libro se supondrá que una no asignada contiene basura. En muchos lenguajes es ilegal usar una variable que contenga basura en una declaración aritmética o desplegarla como salida. Aun si usted trabaja con un lenguaje que le permita desplegar basura, esto no sirve para ningún propósito y constituye un error lógico.

Cuando usted crea una variable sin asignarle un valor inicial (como con `myNumber` y `myAnswer` en la figura 2-2), su intención es asignarlo después; por ejemplo, al recibir uno como entrada o colocar ahí el resultado de un cálculo.

Nombramiento de variables

El ejemplo de la duplicación de números en la figura 2-2 requiere dos variables: `myNumber` y `myAnswer`. De manera alternativa, éstas podrían nombrarse `userEntry` y `programSolution`, o `inputValue` y `twiceTheValue`. Como programador usted elige nombres razonables y descriptivos para sus variables. El intérprete del lenguaje asocia luego los nombres que usted elija con direcciones de memoria específicas.

Cada lenguaje de programación tiene su propio conjunto de reglas para crear identificadores. Casi todos los lenguajes permiten letras y dígitos en los identificadores; algunos aceptan guiones en los nombres de las variables, como `hourly-wage`, y otros permiten subrayados, como en `hourly_wage`. Unos aceptan signos de dólar u otros caracteres especiales (por ejemplo, `hourly$`); otros permiten caracteres de alfabetos extranjeros, como π u Ω. Cada lenguaje tiene algunas (quizá de 100 a 200) **palabras clave** reservadas que no se permite usar como nombres de variables porque son parte de la sintaxis del lenguaje. Cuando usted aprenda un lenguaje de programación conocerá su lista de palabras clave.

Los lenguajes diferentes ponen límites distintos sobre la extensión de los nombres de las variables aunque, en general, la de los identificadores en lenguajes más recientes es casi ilimitada. En muchos lenguajes los identificadores son sensibles al uso de mayúsculas y minúsculas, así `HoUrLyWaGe`, `hourlywage` y `hourlyWage` son tres nombres de variables separados. Los programadores usan numerosas convenciones para nombrar variables, con frecuencia dependiendo del lenguaje de programación o de los estándares implementados por sus patrones. Las convenciones comunes incluyen las siguientes:

- La **notación de camello:** la variable empieza con una letra minúscula y cualquier palabra subsiguiente comienza con una mayúscula, como `hourlyWage`. Para los nombres de las variables en este libro se usa la notación de camello.
- La **caja de Pascal:** la primera letra del nombre de una variable es mayúscula, como en `HourlyWage`.
- La **notación húngara:** el tipo de datos de una variable es parte del identificador; por ejemplo, `numHourlyWage` o `stringLastName`.

Adoptar una convención para nombrar variables y usarla en forma consistente le ayudará para que su programa sea más fácil de leer y entender.

Aun cuando cada lenguaje tiene sus propias reglas para nombrar variables usted no debe preocuparse por la sintaxis específica de alguno en particular cuando diseñe la lógica de un programa. La lógica, después de todo, funciona con cualquier lenguaje. Los nombres de las variables que se usan a lo largo de este libro sólo siguen tres reglas:

1. *Los nombres de las variables deben constar de una palabra.* Pueden contener letras, dígitos, guiones, subrayados o cualquier otro carácter que elija, con excepción de espacios. Por consiguiente, `r` es un nombre de variable legal, lo mismo que `rate` e `interestRate`. El nombre `interest rate` no se permite debido al espacio.

2. *Los nombres de las variables deben comenzar con una letra.* Algunos lenguajes de programación permiten que los nombres comiencen con un carácter no alfabético, como un subrayado. Casi todos los lenguajes no aceptan que los nombres de las variables inicien con un dígito. Este libro sigue la convención más común de empezar los nombres de las variables con una letra.

Cuando usted escribe un programa usando un editor que viene en paquete con un compilador en un IDE, el compilador puede desplegar los nombres de las variables en un color diferente del resto del programa. Esta ayuda visual distingue los nombres de las variables de las palabras que son parte del lenguaje de programación.

3. *Los nombres de las variables deben tener algún significado apropiado.* Ésta no es una regla formal de los lenguajes de programación. Cuando se calcula una tasa de interés en un programa, a la computadora no le interesa si usted llama a la variable `g`, `u84` o `fred`. En tanto que el resultado numérico se coloque en la variable, su nombre real no importa. Sin embargo, es mucho más fácil seguir la lógica de una declaración como `set interestEarned = initialInvestment * interestRate` que de una como `set f = i * r` o `set someBanana = j89 * myFriendLinda`. Cuando un programa requiere cambios, que podrían hacerse meses o años después de que se escribió la versión original, usted y sus colegas programadores apreciarán los nombres descriptivos claros en lugar de los identificadores confusos. En este capítulo usted aprenderá más sobre la selección de identificadores adecuados.

Note que el diagrama de flujo en la figura 2-2 sigue las reglas anteriores: ambos nombres, `myNumber` y `myAnswer`, son palabras únicas sin espacios y tienen significados apropiados. Algunos programadores nombran a las variables en honor a sus amigos o crean juegos de palabras con ellos, pero los profesionales de la computación consideran dicho comportamiento poco profesional y serio.

Asignación de valores a las variables

Cuando deba crear un diagrama de flujo o un seudocódigo para un programa que duplique números puede incluir una declaración como la siguiente:

```
set myAnswer = myNumber * 2
```

Ésta es una **declaración de asignación** e incorpora dos acciones. Primera, la computadora calcula el valor aritmético de `myNumber * 2`. Segunda, el valor calculado se almacena en la ubicación de memoria `myAnswer`.

El signo de igual es el **operador de asignación.** Éste es un ejemplo de **operador binario,** lo que significa que requiere dos operandos, uno en cada lado. El operador de asignación siempre opera de derecha a izquierda, es decir, tiene **asociatividad derecha** o **asociatividad de derecha a izquierda.** Esto significa que se evalúa primero el valor de una expresión a la derecha del operador de asignación, y luego el resultado se asigna al operando de la izquierda. El operando a la derecha de un operador de asignación puede ser un valor, una fórmula, una constante nombrada o una variable. El operando a la izquierda debe ser un nombre que represente una dirección de memoria, el de la ubicación donde se almacenará el resultado.

Por ejemplo, si usted ha declarado dos variables numéricas llamadas `someNumber` y `someOtherNumber`, entonces cada una de las siguientes es una declaración de asignación válida:

```
set someNumber = 2
set someNumber = 3 + 7
set someOtherNumber = someNumber
set someOtherNumber = someNumber * 5
```

En cada caso, la expresión a la derecha del operador de asignación se evalúa y almacena en la ubicación referenciada en el lado izquierdo. El resultado a la izquierda de un operador de asignación se llama un **lvalue.** La *l* es por izquierda en inglés. Los lvalues siempre son identificadores de una dirección de memoria.

Sin embargo, las siguientes declaraciones *no* son válidas:

```
set 2 + 4 = someNumber
set someOtherNumber * 10 = someNumber
```

No lo haga
El operando a la izquierda de un operador de asignación debe representar una dirección de memoria.

En cada uno de estos casos, el valor a la izquierda de un operador de asignación no es una dirección de memoria, de modo que las declaraciones no son válidas.

Cuando usted escriba un seudocódigo o trace un diagrama de flujo sería útil que en las declaraciones de asignación usara la palabra *set*, como se muestra en estos ejemplos, para enfatizar que se establece el valor del lado izquierdo. Sin embargo, en la mayoría de los lenguajes de programación no se usa la palabra *set* y las declaraciones de asignación adoptan la siguiente forma más sencilla:

```
someNumber = 2
someOtherNumber = someNumber
```

Debido a que las asignaciones en la mayoría de los lenguajes aparecen en la forma abreviada, así se usa en el resto de este libro.

Comprensión de los tipos de datos de las variables

Las computadoras manejan los datos de cadena de manera diferente de lo que lo hacen con los datos numéricos. Usted tal vez se ha percatado de estas diferencias si ha usado software de aplicación, como hojas de cálculo o bases de datos. Por ejemplo, en una hoja de cálculo no puede sumar una columna de palabras. Del mismo modo, cada lenguaje de programación requiere que especifique el tipo correcto para cada variable y que use cada tipo de manera apropiada.

- Una **variable numérica** es aquella que puede contener dígitos y operaciones matemáticas que se efectúan en ellos. En este libro todas las variables numéricas pueden contener un punto decimal y un signo que indica positivo o negativo; algunos lenguajes de programación proporcionan tipos numéricos especializados para estas opciones. En la declaración `myAnswer = myNumber * 2`, tanto `myAnswer` como `myNumber` son variables numéricas; es decir, su contenido previsto son valores numéricos, como 6 y 3, 14.8 y 7.4 o –18 y –9.
- Una **variable de cadena** puede contener texto, como las letras del alfabeto y otros caracteres especiales, como los signos de puntuación. Si un programa en funcionamiento contiene la declaración `lastName = "Lincoln"`, entonces `lastName` es una variable de cadena. Una

variable de cadena también puede contener dígitos ya sea con o sin otros caracteres. Por ejemplo, tanto "235 Main Street" como "86" son cadenas. Una cadena como "86" se almacena de manera diferente que el valor numérico 86 y usted no puede realizar aritmética con la cadena.

La **seguridad del tipo** es la característica de los lenguajes de programación que impide asignar valores de un tipo de datos incorrecto. Usted puede asignar datos a una variable sólo si son del tipo correcto. Si usted declara `taxRate` como una variable numérica e `inventoryItem` como una cadena, entonces las siguientes declaraciones son válidas:

```
taxRate = 2.5
inventoryItem = "monitor"
```

Las siguientes no son válidas debido a que el tipo de datos que se asigna no corresponden al tipo de la variable:

```
taxRate = "2.5"
inventoryItem = 2.5
```

No lo haga

Si taxRate es numérico e inventoryItem es una cadena, entonces estas asignaciones no son válidas.

Declaración de constantes nombradas

Además de las variables, casi todos los lenguajes de programación permiten la creación de constantes nombradas. Una **constante nombrada** es similar a una variable, excepto que se le puede asignar un valor sólo una vez. Se usa cuando se desea asignar un nombre útil para un valor que nunca cambiará durante la ejecución de un programa. El uso de constantes nombradas hace que sus programas sean más fáciles de entender al eliminar números mágicos. Un **número mágico** es una constante literal, como 0.06, cuyo propósito no es evidente de inmediato.

Por ejemplo, si un programa usa una tasa de impuesto sobre las ventas de 6%, quizá desee declarar una constante nombrada como sigue:

```
num SALES_TAX_RATE = 0.06
```

Ya que se ha declarado `SALES_TAX_RATE`, las siguientes declaraciones tienen significado idéntico:

```
taxAmount = price * 0.06
taxAmount = price * SALES_TAX_RATE
```

La forma en que se declaran las constantes nombradas difiere entre los lenguajes de programación. Este libro sigue la convención de usar sólo letras mayúsculas en los identificadores de las constantes y subrayados para separar las palabras por legibilidad. Estas normas facilitan el reconocimiento de las constantes nombradas. En muchos lenguajes debe asignarse un valor a una constante cuando se declara, pero en otros es posible hacerlo después. Sin embargo, en ningún caso es posible cambiar el valor de una constante después de la primera asignación. Aquí se inicializan todas las constantes cuando se declaran.

Cuando usted declara una constante nombrada, el mantenimiento del programa se vuelve más fácil. Por ejemplo, si el valor del impuesto sobre las ventas cambia de 0.06 a 0.07 en el futuro, y usted ha declarado una constante llamada `SALES_TAX_RATE`, sólo necesita cambiar el valor asignado a la constante nombrada al principio del programa, luego retraducir éste

al lenguaje de máquina y todas las referencias a SALES_TAX_RATE se actualizan en forma automática. Si en su lugar usa la constante literal 0.06, tendría que buscar cada caso del valor y reemplazarlo con el nuevo. Además, si la literal 0.06 se usó en otros cálculos dentro del programa (por ejemplo, como tasa de descuento), tendría que seleccionar con cuidado cuáles casos del valor se alterarán y quizá cometa un error.

En ocasiones, el uso de constantes literales es apropiado en un programa, en especial si su significado es claro para la mayoría de los lectores. Por ejemplo, en un programa que calcula la mitad de un valor dividiéndolo entre dos, podría elegir la constante literal 2 en lugar de incurrir en el tiempo y los costos de memoria adicionales de crear una constante nombrada HALF y asignarle 2. Los costos adicionales que resulten de agregar variables o instrucciones al programa se conocen como **carga adicional.**

DOS VERDADES Y UNA MENTIRA

Declaración y uso de variables y constantes

1. El tipo de datos de una variable describe la clase de valores que ésta puede contener y los tipos de operaciones que es posible efectuar con ellos.
2. Si `name` es una variable de cadena, entonces la declaración `set name = "Ed"` es válida.
3. El operando a la derecha de un operador de asignación debe ser un nombre que represente una dirección de memoria.

La afirmación falsa es la número 3. El operando a la izquierda de un operador de asignación debe ser un nombre que represente una dirección de memoria; el nombre de la ubicación donde se almacenará el resultado. Cualquier operando a la derecha de un operador de asignación puede ser una dirección de memoria (una variable) o una constante.

Realización de operaciones aritméticas

La mayoría de los lenguajes de programación usan los siguientes operadores aritméticos estándar:

+ (signo de suma): adición

– (signo de resta): sustracción

* (asterisco): multiplicación

/ (diagonal): división

Muchos lenguajes también soportan operadores adicionales que calculan el residuo después de la división, elevan un número a una potencia mayor, manipulan bits individuales almacenados dentro de un valor y efectúan otras operaciones.

Cada uno de los operadores aritméticos estándar es un operador binario; es decir, que requiere una expresión en ambos lados. Por ejemplo, la siguiente declaración suma dos puntuaciones de examen y asigna la suma a una variable llamada `totalScore`:

```
totalScore = test1 + test2
```

La siguiente suma 10 a `totalScore` y almacena el resultado en `totalScore`:

```
totalScore = totalScore + 10
```

En otras palabras, este ejemplo aumenta el valor de `totalScore`. Este último ejemplo se ve extraño en álgebra porque parecería que el valor de `totalScore` y el de `totalScore` más 10 son equivalentes. Debe recordar que el signo de igual es el operador de asignación y que la declaración en realidad toma el valor original de `totalScore`, sumándole 10 y asignando el resultado a la dirección de memoria a la izquierda del operador, que es `totalScore`.

En los lenguajes de programación usted puede combinar declaraciones aritméticas. Cuando lo hace, cada operador sigue las **reglas de precedencia** (también llamadas **orden de las operaciones**) que dictan el orden en que se realizan las operaciones en la misma declaración. Las reglas de precedencia para las declaraciones aritméticas básicas son las siguientes:

- Las expresiones entre paréntesis se evalúan primero. Si hay varios conjuntos de paréntesis, la expresión dentro del más interior se evalúa primero.
- La multiplicación y la división se evalúan a continuación, de izquierda a derecha.
- La suma y la resta se evalúan después, de izquierda a derecha.

El operador de asignación tiene poca precedencia. Por consiguiente, en una declaración como `d = e * f + g`, las operaciones a la derecha del operador de asignación siempre se efectúan antes de la asignación final a la variable en la izquierda.

Cuando usted aprenda un lenguaje de programación específico, conocerá todos los operadores que se usan en él. Muchos libros sobre el tema contienen una tabla que especifica la precedencia relativa de todos los operadores que se usan en el lenguaje.

Por ejemplo, considere las siguientes dos declaraciones aritméticas:

```
firstAnswer = 2 + 3 * 4
secondAnswer = (2 + 3) * 4
```

Después de que se ejecutan estas declaraciones, el valor de `firstAnswer` es 14. De acuerdo con las reglas de precedencia, la multiplicación se efectúa antes que la suma, así que 3 se multiplica por 4 y resulta 12, y luego se suman 2 y 12, y se asigna 14 a `firstAnswer`. Sin embargo, el valor de `secondAnswer` es 20, porque el paréntesis obliga a que se efectúe primero la operación de suma que contiene. Se suman 2 y 3, lo que resulta 5, y luego 5 se multiplica por 4, lo que da 20.

El olvido de las reglas de precedencia aritmética o de añadir los paréntesis cuando se necesitan puede generar errores lógicos que son difíciles de encontrar en los programas. Por ejemplo, la siguiente declaración parecería promediar dos puntuaciones de examen:

```
average = score1 + score2 / 2
```

Sin embargo, no es así. Debido a que la división tiene una precedencia mayor que la suma, la declaración anterior indica que se obtiene la mitad de `score2`, se suma a `score1` y el resultado se almacena en `average`. La declaración correcta es:

```
average = (score1 + score2) / 2
```

Usted es libre de agregar paréntesis aun cuando no los necesite para forzar un orden diferente de operaciones; en ocasiones se usan sólo para hacer más claras sus intenciones. Por ejemplo, las siguientes declaraciones operan en forma idéntica:

```
totalPriceWithTax = price + price * TAX_RATE
totalPriceWithTax = price + (price * TAX_RATE)
```

En ambos casos, `price` se multiplica primero por `TAX_RATE`, luego este resultado se suma a `price` y lo que se obtiene al final se almacena en `totalPriceWithTax`. Debido a que la multiplicación ocurre antes de la suma en el lado derecho del operador de asignación, ambas declaraciones son iguales. Sin embargo, si usted piensa que la declaración con paréntesis hace más claras sus intenciones para alguien que lea su programa, entonces debe usarlos.

Todos los operadores aritméticos tienen **asociatividad de izquierda a derecha.** Esto significa que las operaciones con la misma precedencia tienen lugar de izquierda a derecha. Considere la siguiente declaración:

```
answer = a + b + c * d / e – f
```

La multiplicación y la división tienen una precedencia mayor que la suma o la resta, así que se llevan a cabo de izquierda a derecha como sigue:

`c` se multiplica por `d`, y el resultado se divide entre `e`, lo que da un resultado nuevo.

Por tanto, la declaración queda:

```
answer = a + b + (resultado temporal que ya se ha calculado) – f
```

Entonces se efectúan la suma y la resta de izquierda a derecha como sigue:

Se suman `a` y `b`, se suma el resultado temporal, y luego se resta `f`. El resultado final se asigna entonces a `answer`.

Otra forma de decirlo es que las siguientes dos declaraciones son equivalentes:

```
answer = a + b + c * d / e – f
answer = a + b + ((c * d) / e) – f
```

El cuadro 2-1 resume la precedencia y asociatividad de los cinco operadores que se usan con mayor frecuencia.

Símbolo del operador	Nombre del operador	Precedencia (comparada con otros operadores en este cuadro)	Asociatividad
=	Asignación	Más baja	Derecha a izquierda
+	Suma	Media	Izquierda a derecha
–	Resta	Media	Izquierda a derecha
*	Multiplicación	Más alta	Izquierda a derecha
/	División	Más alta	Izquierda a derecha

Cuadro 2-1 Precedencia y asociatividad de cinco operadores comunes

DOS VERDADES Y UNA MENTIRA

Realización de operaciones aritméticas

1. Los paréntesis tienen mayor precedencia que cualquiera de los operadores aritméticos comunes.
2. Las operaciones en declaraciones aritméticas ocurren de izquierda a derecha en el orden en que aparecen.
3. La siguiente suma 5 a una variable nombrada `points`:

```
points = points + 5
```

La afirmación falsa es la número 2. Las operaciones de igual precedencia en una declaración aritmética se efectúan de izquierda a derecha, pero las operaciones que se encuentran entre paréntesis se realizan primero, la multiplicación y la división se efectúan a continuación, y la suma y la resta se hacen al último.

Comprensión de las ventajas de la modularización

Los programadores rara vez escriben los programas como una larga serie de pasos. En cambio, dividen sus problemas de programación en unidades más pequeñas y abordan una tarea cohesiva a la vez. Estas unidades más pequeñas son **módulos.** Los programadores también se refieren a ellos como **subrutinas, procedimientos, funciones** o **métodos;** el nombre por lo general refleja el lenguaje de programación que se esté usando. Por ejemplo, los programadores en Visual Basic usan *procedimiento* (o *subprocedimiento*). En C y C++ los módulos se llaman *funciones,* mientras que en C#, Java y otros lenguajes orientados hacia los objetos es más probable que se conozcan como *método.* En COBOL, RPG y BASIC (lenguajes más antiguos) en general se denominan *subrutinas.*

Un programa principal ejecuta un módulo al llamarlo. **Llamar a un módulo** quiere decir que se usa su nombre para atraerlo, causando que se ejecute. Cuando las tareas del módulo están completas, el control regresa al punto desde el que se llamó en el programa principal. Cuando usted entra a un módulo, la acción es parecida a la de poner en pausa un reproductor de DVD: abandona su acción primaria (ver un video), se ocupa de alguna otra tarea (por ejemplo, hacer un emparedado) y luego regresa a la tarea principal exactamente donde la dejó.

La **modularización** es el proceso de descomponer un programa extenso en módulos; los científicos en computación también la llaman **descomposición funcional.** Nunca se requiere que modularice un programa extenso para que corra en una computadora, pero hay al menos tres razones para hacerlo:

- La modularización proporciona abstracción.
- Permite que muchos programadores trabajen en un problema.
- Hace posible reutilizar el trabajo con más facilidad.

La modularización proporciona abstracción

Una razón para que sea más fácil entender los programas modularizados es que permiten a un programador ver "todo el panorama". La **abstracción** es el proceso de poner atención en las propiedades importantes mientras se ignoran los detalles no esenciales. La abstracción es ignorancia selectiva; la vida sería tediosa sin ella. Por ejemplo, usted puede crear una lista de cosas que hará hoy:

```
Lavar ropa
Llamar a la tía Nan
Empezar el ensayo semestral
```

Sin abstracción, la lista de quehaceres comenzaría así:

```
Levantar el cesto de ropa sucia
Poner el cesto de ropa sucia en el automóvil
Conducir hasta la lavandería
Salir del automóvil con el cesto
Entrar a la lavandería
Bajar el cesto
Encontrar monedas para la máquina lavadora
... etcétera.
```

Usted podría enumerar una docena de pasos más antes de terminar de lavar la ropa y pasar a la segunda labor en su lista original. Si tuviera que considerar todos los detalles pequeños de bajo nivel de cada tarea en su día es probable que nunca saliera de la cama por la mañana. Cuando usa una lista más abstracta de nivel superior hace su día más manejable. La abstracción hace que las tareas complejas se vean más sencillas.

Los artistas abstractos crean pinturas en las que sólo se ve el panorama general (color y forma) y se ignoran los detalles. La abstracción tiene un significado similar entre los programadores.

Del mismo modo, ocurre algún nivel de abstracción en cada programa de computadora. Hace 50 años un programador tenía que entender las instrucciones de circuitería de bajo nivel que la máquina usaba, pero ahora los lenguajes de programación de alto nivel más recientes permiten el uso de vocabulario en inglés en el que una declaración amplia corresponde a docenas de instrucciones. Sin importar cuál lenguaje de programación de alto nivel use, cuando usted despliega un mensaje en el monitor nunca se le pide que entienda cómo funciona éste para crear cada píxel en la pantalla. Usted escribe una instrucción como `output message` y el sistema operativo maneja los detalles de las operaciones de hardware para usted.

Los módulos brindan otra opción para lograr la abstracción. Por ejemplo, un programa de nómina puede llamar a un módulo que se ha nombrado `computeFederalWithholdingTax()`. Cuando usted lo llama desde su programa usa una declaración; el módulo en sí podría contener docenas de declaraciones. Usted puede escribir los detalles matemáticos del módulo después, alguien más puede hacerlo o puede adquirirlos de una fuente externa. Cuando usted planea su programa de nómina principal, su única preocupación es que tendrá que calcularse la retención de un impuesto federal; guarde los detalles para más tarde.

La modularización permite que varios programadores trabajen en un problema

Cuando usted disecciona cualquier tarea grande en módulos, tiene la capacidad de dividir con más facilidad la tarea entre varias personas. Rara vez un solo programador escribe un programa comercial que se pueda comprar. Considere cualquier procesador de palabras, hoja de cálculo o base de datos que haya usado. Cada programa tiene tantas opciones y responde a las selecciones del usuario en tantas formas posibles, que tomaría años que un solo programador escribiera todas las instrucciones. Los desarrolladores de software profesionales pueden escribir programas nuevos en semanas o meses, en lugar de años, si dividen los programas extensos en módulos y asignan cada uno a un programador individual o un equipo.

La modularización permite que se reutilice el trabajo

Si un módulo es útil y está bien escrito quizá usted desee usarlo más de una vez en ese programa o en otros. Por ejemplo, una rutina que verifica la validez de las fechas es útil en muchos programas escritos para un negocio. (Por ejemplo, un valor de mes es válido si no es menor que 1 o mayor que 12, un valor de día es válido si no es menor que 1 o mayor que 31 si el mes es 1, etc.). Si un archivo de personal computarizado contiene las fechas de nacimiento, de contratación, del último ascenso y de renuncia de cada empleado, el módulo de validación de fechas puede usarse cuatro veces con cada registro de empleado. Otros programas en una organización también pueden usar el módulo; podrían embarcar pedidos del cliente, planear fiestas de cumpleaños de los empleados o calcular cuándo deben hacerse los pagos de un préstamo. Si usted escribe las instrucciones de comprobación de las fechas de modo que estén entremezcladas con otras declaraciones en un programa, extraerlas y reutilizarlas será difícil. Por otra parte, si coloca las instrucciones en su propio módulo, será fácil usar y transportar la unidad a otras aplicaciones. La característica de los programas modulares que permite usar módulos individuales en una variedad de aplicaciones es la **reutilización.**

Es posible encontrar muchos ejemplos de reutilización en el mundo real. Cuando alguien construye una casa no inventa la plomería y los sistemas de calefacción; incorpora sistemas con diseños probados. Esto reduce con seguridad el tiempo y el esfuerzo que se requieren para construirla. Los sistemas de plomería y eléctricos que se elijan están en servicio en otras casas, así que han sido probados en diversas circunstancias, lo que aumenta su confiabilidad. La **confiabilidad** es la característica de los programas que asegura que un módulo funciona en forma correcta. El software confiable ahorra tiempo y dinero. Si usted crea los componentes funcionales de sus programas como módulos independientes y los prueba en sus programas actuales, gran parte del trabajo estará hecho cuando use los módulos en aplicaciones futuras.

DOS VERDADES Y UNA MENTIRA

Comprensión de las ventajas de la modularización

1. La modularización elimina la abstracción, una característica que hace que los programas sean confusos.
2. La modularización hace más fácil que muchos programadores trabajen en un problema.
3. La modularización permite reutilizar el trabajo con más facilidad.

La afirmación falsa es la número 1. La modularización permite la abstracción, lo que permite ver el panorama general.

Modularización de un programa

La mayoría de los programas consiste en un **programa principal,** que contiene los pasos básicos, o la **lógica de línea principal** del programa. Entonces el programa principal entra en los módulos que proporcionan detalles más refinados.

Cuando se crea un módulo se incluye lo siguiente:

- Un encabezado: el **encabezado del módulo** incluye el identificador del mismo y posiblemente otra información de identificación necesaria.
- Un cuerpo: El **cuerpo del módulo** contiene todas las declaraciones en el mismo.
- Una declaración `return`: La **declaración `return` del módulo** marca el final de éste e identifica el punto en que el control regresa al programa o módulo que llamó al otro. En casi todos los lenguajes de programación, si no se incluye una declaración `return` al final de un módulo, la lógica todavía regresará. Sin embargo, este libro sigue la norma de incluir en forma explícita una declaración `return` con cada módulo.

Nombrar un módulo es algo similar a nombrar una variable. Las reglas para hacerlo son ligeramente diferentes en cada lenguaje de programación, pero en este texto los nombres de los módulos siguen las mismas reglas generales que se usan para los identificadores de variables:

- Los nombres de los módulos deben constar de una palabra y comenzar con una letra.
- Los nombres de los módulos deben tener algún significado.

Introducción
A la Programación Lógica y Diseño
7a. Ed. | JOYCE FARRELL

Prepare para el éxito a los programadores principiantes con la muy efectiva *Introducción a la Programación Lógica y Diseño*, de Farrell. Este texto popular adopta un enfoque único independiente del lenguaje de programación con un énfasis en las convenciones modernas. El estilo de redacción del libro, claro y conciso, elimina la jerga altamente técnica mientras presenta conceptos universales de programación y alienta un estilo de programación y pensamiento lógico sólidos. Las explicaciones revisadas y más definidas de esta edición incorporan diagramas de flujo, seudocódigo y diagramas para asegurar que incluso los lectores sin experiencia previa en programación entiendan por completo los conceptos de la programación y el diseño modernos.

CARACTERÍSTICAS DEL TEXTO

- Ejercicios adicionales basados en la elaboración de diagramas de flujo y seudocódigo en esta edición ayudan a los estudiantes a obtener una mejor comprensión del alcance de la programación moderna.
- Explicaciones revisadas minuciosamente proporcionan la guía más clara posible para los lectores que no tienen experiencia previa en programación.
- Una abundancia de oportunidades de práctica probadas, incluyendo Ejercicios de programación, Ejercicios de eliminación de errores y cuestionarios "Dos verdades y una mentira", mantienen a los estudiantes interesados y aprendiendo en forma activa.

ACERCA DEL AUTOR

Joyce Farrell ha escrito una amplia variedad de libros de texto de programación exitosos reconocidos por su estilo de redacción directo y claro y su presentación efectiva. Es autora de varios textos de Course Technology, incluyendo versiones orientadas hacia los objetos y extendidas de este libro, al igual que *Java Programming, Microsoft Visual C++* y *Object-Oriented Programming Using C++*. Instructora muy respetada, Farrell impartió temas de sistemas de información de computadora por más de 20 años en colegios en Illinois y Wisconsin.

CENGAGE Learning®

Visite nuestro sitio en http://latinoamerica.cengage.com

ISBN-13: 978-607481904-5
ISBN-10: 607481904-1

9 786074 819045