15

Recursividad

Debemos aprender a explorar todas las opciones y posibilidades a las que nos enfrentamos en un mundo complejo, que evoluciona rápidamente.

—James William Fulbright

Oh, maldita iteración, que eres capaz de corromper hasta a un santo.

—William Shakespeare

Es un pobre orden de memoria, que sólo funciona al revés.

—Lewis Carroll

La vida sólo puede comprenderse al revés; pero debe vivirse hacia delante.

—Soren Kierkegaard

Empujen; sigan avanzando.

—Thomas Morton

OBJETIVOS

En este capítulo aprenderá a:

- Comprender el concepto de recursividad.
- Escribir y utilizar métodos recursivos.
- Determinar el caso base y el paso de recursividad en un algoritmo recursivo.
- Conocer cómo el sistema maneja las llamadas a métodos recursivos.
- Conocer las diferencias entre recursividad e iteración, y cuándo es apropiado utilizar cada una.
- Conocer las figuras geométricas llamadas fractales, y cómo se dibujan mediante la recursividad.
- Conocer el concepto de "vuelta atrás" recursiva (backtracking), y por qué es una técnica efectiva para solucionar problemas.

Plan general

15.1 Introducción

Los programas que hemos visto hasta ahora están estructurados generalmente como métodos que se llaman entre sí, de una manera disciplinada y jerárquica. Sin embargo, para algunos problemas es conveniente hacer que un método se llame a sí mismo. Dicho método se conoce como **método recursivo**; este método se puede llamar en forma directa o indirecta a través de otro método. La recursividad es un tema importante, que puede tratarse de manera extensa en los cursos de ciencias computacionales de nivel superior. En este capítulo consideraremos la recursividad en forma conceptual, y después presentaremos varios programas que contienen métodos recursivos. En la figura 15.1 se sintetizan los ejemplos y ejercicios de recursividad que se incluyen en este libro.

Capítulo	Ejemplos y ejercicios de recursividad en este libro
15	Método factorial (figuras 15.3 y 15.4.) Método Fibonacci (figuras 15.5 y 15.6). Torres de Hanoi (figuras 15.13 y 15.14). Fractales (figuras 15.21 y 15.22). ¿Qué hace este código? (ejercicios 15.7, 15.12 y 15.13). Encuentre el error en el siguiente código (ejercicio 15.8). Elevar un entero a una potencia entera (ejercicio 15.9). Visualización de la recursividad (ejercicio 15.10). Máximo común divisor (ejercicio 15.11). Determinar si una cadena es un palíndromo (ejercicio 15.14). Ocho reinas (ejercicio 15.15). Imprimir un arreglo (ejercicio 15.16). Imprimir un arreglo al revés (ejercicio 15.17). Mínimo valor en un arreglo (ejercicio 15.18). Fractal de estrella (ejercicio 15.19). Recorrido de un laberinto mediante el uso de la "vuelta atrás" recursiva (ejercicio 15.20). Generación de laberintos al azar (ejercicio 15.21). Laberintos de cualquier tamaño (ejercicio 15.22). Tiempo para calcular números de Fibonacci (ejercicio 15.23).
16	Ordenamiento por combinación (figuras 16.10 y 16.11). Búsqueda lineal recursiva (ejercicio 16.8). Búsqueda binaria recursiva (ejercicio 16.9). Quicksort (ejercicio 16.10).

Figura 15.1 | Resumen de los ejemplos y ejercicios de recursividad en este libro. (Parte 1 de 2).

Capítulo	Ejemplos y ejercicios de recursividad en este libro
17	Inserción en árbol binario (figura 17.17). Recorrido preorden de un árbol binario (figura 17.17). Recorrido inorden de un árbol binario (figura 17.17). Recorrido postorden de un árbol binario (figura 17.17). Imprimir una lista en forma recursiva y en forma inversa (ejercicio 17.20). Buscar en una lista en forma recursiva (ejercicio 17.21).

Figura 15.1 | Resumen de los ejemplos y ejercicios de recursividad en este libro. (Parte 2 de 2).

15.2 Conceptos de recursividad

Los métodos para solucionar problemas recursivos tienen varios elementos en común. Cuando se hace una llamada a un método recursivo para resolver un problema, el método en realidad es capaz de resolver sólo el (los) caso(s) más simple(s), o **caso(s) base**. Si se hace la llamada al método con un caso base, el método devuelve un resultado. Si se hace la llamada al método con un problema más complejo, el método comúnmente divide el problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver y otra pieza que no sabe cómo resolver. Para que la recursividad sea factible, esta última pieza debe ser similar al problema original, pero una versión ligeramente más sencilla o simple del mismo. Debido a que este nuevo problema se parece al problema original, el método llama a una nueva copia de sí mismo para trabajar en el problema más pequeño; a esto se le conoce como **llamada recursiva**, y también como **paso recursivo**. Por lo general, el paso recursivo incluye una instrucción `return`, ya que su resultado se combina con la parte del problema que el método supo cómo resolver, para formar un resultado que se pasará de vuelta al método original que hizo la llamada. Este concepto de separar el problema en dos porciones más pequeñas es una forma del método "divide y vencerás" que presentamos en el capítulo 6.

El paso recursivo se ejecuta mientras siga activa la llamada original al método (es decir, que no haya terminado su ejecución). Se pueden producir muchas llamadas recursivas más, a medida que el método divide cada nuevo subproblema en dos piezas conceptuales. Para que la recursividad termine en un momento dado, cada vez que el método se llama a sí mismo con una versión más simple del problema original, la secuencia de problemas cada vez más pequeños debe converger en un caso base. En ese punto, el método reconoce el caso base y devuelve un resultado a la copia anterior del método. Después se origina una secuencia de retornos, hasta que la llamada al método original devuelve el resultado final al método que lo llamó.

Un método recursivo puede llamar a otro método, que a su vez puede hacer una llamada de vuelta al método recursivo. A dicho proceso se le conoce como **llamada recursiva indirecta** o **recursividad indirecta**. Por ejemplo, el método `A` llama al método `B`, que hace una llamada de vuelta al método `A`. Esto se sigue considerando como recursividad, debido a que la segunda llamada al método `A` se realiza mientras la primera sigue activa; es decir, la primera llamada al método `A` no ha terminado todavía de ejecutarse (debido a que está esperando que el método `B` le devuelva un resultado) y no ha regresado al método original que llamó al método `A`.

Para comprender mejor el concepto de recursividad, veamos un ejemplo que es bastante común para los usuarios de computadora: la definición recursiva de un directorio en una computadora. Por lo general, una computadora almacena los archivos relacionados en un directorio. Este directorio puede estar vacío, puede contener archivos y/o puede contener otros directorios (que, por lo general, se conocen como subdirectorios). A su vez, cada uno de estos directorios puede contener también archivos y directorios. Si queremos enlistar cada archivo en un directorio (incluyendo todos los archivos en los subdirectorios de ese directorio), necesitamos crear un método que lea primero los archivos del directorio inicial y que después haga llamadas recursivas para enlistar los archivos en cada uno de los subdirectorios de ese directorio. El caso base ocurre cuando se llega a un directorio que no contenga subdirectorios. En este punto, se han enlistado todos los archivos en el directorio original y no se necesita más la recursividad.

15.3 Ejemplo de uso de recursividad: factoriales

Escribimos un programa recursivo, para realizar un popular cálculo matemático. Considere el factorial de un entero positivo *n*, escrito como *n*! (y se pronuncia como "factorial de *n*"), que viene siendo el producto

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

en donde 1! es igual a 1 y 0! se define como 1. Por ejemplo, 5! es el producto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es igual a 120.

El factorial del entero `numero` (en donde `numero ≥ 0`) puede calcularse de manera **iterativa** (sin recursividad), usando una instrucción `for` de la siguiente manera:

```
factorial = 1;

for ( int contador = numero; contador >= 1; contador-- )
    factorial *= contador;
```

Podemos llegar a una declaración recursiva del método del factorial, si observamos la siguiente relación:

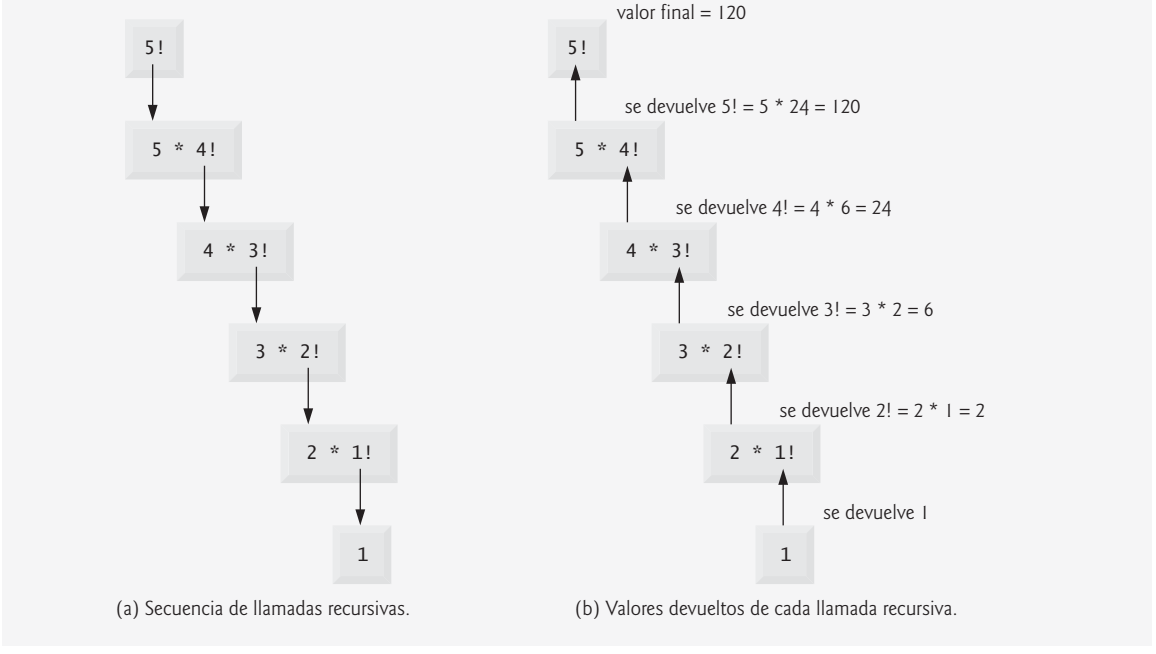
$$n! = n \cdot (n-1)!$$

Por ejemplo, 5! es sin duda igual a $5 \cdot 4!$, como se muestra en las siguientes ecuaciones:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

La evaluación de 5! procedería como se muestra en la figura 15.2. La figura 15.2(a) muestra cómo procede la sucesión de llamadas recursivas hasta que 1! (el caso base) se evalúa como 1, lo cual termina la recursividad. La figura 15.2(b) muestra los valores devueltos de cada llamada recursiva al método que hizo la llamada, hasta que se calcula y devuelve el valor final.

En la figura 15.3 se utiliza la recursividad para calcular e imprimir los factoriales de los enteros del 0 al 10. El método recursivo `factorial` (líneas 7 a 13) realiza primero una evaluación para determinar si una condición de terminación (línea 9) es `true`. Si `numero` es menor o igual que `1` (el caso base), `factorial` devuelve `1`, ya no es necesaria más recursividad y el método regresa. Si `numero` es mayor que `1`, en la línea 12 se expresa el problema como el producto de `numero` y una llamada recursiva a `factorial` en la que se evalúa el factorial de `numero - 1`, el cual es un problema un poco más pequeño que el cálculo original, `factorial( numero )`.



(a) Secuencia de llamadas recursivas.

(b) Valores devueltos de cada llamada recursiva.

Figura 15.2 | Evaluación recursiva de 5!.

```
1  // Fig. 15.3: CalculoFactorial.java
2  // Método factorial recursivo.
3
4  public class CalculoFactorial
5  {
6     // declaración recursiva del método factorial
7     public long factorial( long numero )
8     {
9        if ( numero <= 1 ) // evalúa el caso base
10          return 1; // casos base: 0! = 1 y 1! = 1
11       else // paso recursivo
12          return numero * factorial( numero - 1 );
13    } // fin del método factorial
14
15    // imprime factoriales para los valores del 0 al 10
16    public void mostrarFactoriales()
17    {
18       // calcula los factoriales del 0 al 10
19       for ( int contador = 0; contador <= 10; contador++ )
20          System.out.printf( "%d! = %d\n", contador, factorial( contador ) );
21    } // fin del método mostrarFactoriales
22 } // fin de la clase CalculoFactorial
```

Figura 15.3 | Cálculos de factoriales con un método recursivo.

Error común de programación 15.1

Si omitimos el caso base o escribimos el paso recursivo en forma incorrecta, de manera que no converja en el caso base, se puede producir un error lógico conocido como ***recursividad infinita****, en donde se realizan llamadas recursivas en forma continua, hasta que se agota la memoria. Este error es análogo al problema de un ciclo infinito en una solución iterativa (sin recursividad).*

El método `mostrarFactoriales` (líneas 16 a 21) muestra los factoriales del 0 al 10. La llamada al método `factorial` ocurre en la línea 20. Este método recibe un parámetro de tipo `long` y devuelve un resultado de tipo `long`. La figura 15.4 prueba nuestros métodos `factorial` y `mostrarFactoriales`, llamando a `mostrarFactoriales` (línea 10). Los resultados de la figura 15.4 muestra que los valores de los factoriales crecen rápidamente. Utilizamos el tipo `long` (que puede representar enteros relativamente grandes) para que el programa pueda calcular factoriales mayores que 12!. Por desgracia, el método `factorial` produce valores grandes con tanta rapidez que los valores de los factoriales exceden pronto al valor máximo que puede almacenarse, incluso en una variable `long`.

Debido a las limitaciones de los tipos integrales, tal vez se necesiten variables `float` o `double` para calcular factoriales o números grandes. Esto apunta a una debilidad en la mayoría de los lenguajes de programación: a saber, que los lenguajes no se extienden fácilmente para manejar los requerimientos únicos de una aplicación. Como vimos en el capítulo 9, Java es un lenguaje extensible que nos permite crear números arbitrariamente grandes, si lo deseamos. De hecho, el paquete `java.math` cuenta con las clases `BigInteger` y `BigDecimal` explícitamente para los cálculos matemáticos de precisión arbitraria, que no pueden llevarse a cabo con los tipos primitivos. Para obtener más información acerca de estas clases, visite `java.sun.com/javase/6/docs/api/java/math/BigInteger.html` y `java.sun.com/javase/6/docs/api/java/math/BigDecimal.html`, respectivamente.

```
1  // Fig. 15.4: PruebaFactorial.java
2  // Prueba del método recursivo factorial.
3
4  public class PruebaFactorial
```

Figura 15.4 | Prueba del método `factorial`. (Parte 1 de 2).

```
5  {
6     // calcula los factoriales del 0 al 10
7     public static void main( String args[] )
8     {
9        CalculoFactorial calculoFactorial = new CalculoFactorial();
10       calculoFactorial.mostrarFactoriales();
11    } // fin del método main
12 } // fin de la clase PruebaFactorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Figura 15.4 | Prueba del método `factorial`. (Parte 2 de 2).

15.4 Ejemplo de uso de recursividad: serie de Fibonacci

La **serie de Fibonacci**,

0, 1, 1, 2, 3, 5, 8, 13, 21, …

empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos números Fibonacci anteriores. Esta serie ocurre en la naturaleza y describe una forma de espiral. La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número denominado **proporción dorada**, o **media dorada**. Los humanos tienden a descubrir que la media dorada es estéticamente placentera. A menudo, los arquitectos diseñan ventanas, cuartos y edificios con una proporción de longitud-anchura en la que se utiliza la media dorada.

La serie de Fibonacci se puede definir de manera recursiva como:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Observe que hay dos casos base para el cálculo de Fibonacci: `fibonacci(0)` se define como 0, y `fibonacci(1)` se define como 1. El programa de la figura 15.5 calcula el i-ésimo número de Fibonacci en forma recursiva, usando el método `fibonacci` (líneas 7 a 13). El método `mostrarFibonacci` (líneas 15 a 20) prueba a `fibonacci`, mostrando los valores de Fibonacci del 0 al 10. La variable `contador` creada en el encabezado de la instrucción `for` en la línea 17 indica cuál número de Fibonacci se debe calcular para cada iteración del número `for`. Los números de Fibonacci tienden a aumentar con rapidez. Por lo tanto, utilizamos `long` como el tipo del parámetro y el tipo de valor de retorno del método `fibonacci`. En la línea 9 de la figura 15.6 se hace una llamada al método `mostrarFibonacci` (línea 9) para calcular los valores de Fibonacci.

```
1  // Fig. 15.5: CalculoFibonacci.java
2  // Método fibonacci recursivo.
3
4  public class CalculoFibonacci
5  {
```

Figura 15.5 | Números de `Fibonacci` generados con un método recursivo. (Parte 1 de 2).

```
6     // declaración recursiva del método fibonacci
7     public long fibonacci( long numero )
8     {
9        if ( ( numero == 0 ) || ( numero == 1 ) ) // casos base
10          return numero;
11       else // paso recursivo
12          return fibonacci( numero - 1 ) + fibonacci( numero - 2 );
13    } // fin del método fibonacci
14
15    public void mostrarFibonacci()
16    {
17       for ( int contador = 0; contador <= 10; contador++ )
18          System.out.printf( "Fibonacci de %d es: %d\n", contador,
19             fibonacci( contador ) );
20    } // fin del método mostrarFibonacci
21 } // fin de la clase CalculoFibonacci
```

Figura 15.5 | Números de `Fibonacci` generados con un método recursivo. (Parte 2 de 2).

```
1  // Fig. 15.6: PruebaFibonacci.java
2  // Prueba del método recursivo fibonacci.
3
4  public class PruebaFibonacci
5  {
6     public static void main( String args[] )
7     {
8        CalculoFibonacci calculoFibonacci = new CalculoFibonacci();
9        calculoFibonacci.mostrarFibonacci();
10    } // fin de main
11 } // fin de la clase PruebaFibonacci
```

```
Fibonacci de 0 es: 0
Fibonacci de 1 es: 1
Fibonacci de 2 es: 1
Fibonacci de 3 es: 2
Fibonacci de 4 es: 3
Fibonacci de 5 es: 5
Fibonacci de 6 es: 8
Fibonacci de 7 es: 13
Fibonacci de 8 es: 21
Fibonacci de 9 es: 34
Fibonacci de 10 es: 55
```

Figura 15.6 | Prueba del método `Fibonacci`.

La llamada al método `fibonacci` (línea 19 de la figura 15.5) desde `mostrarFibonacci` no es una llamada recursiva, pero todas las llamadas subsiguientes a `fibonacci` que se llevan a cabo desde el cuerpo de `fibonacci` (línea 12 de la figura 15.5) son recursivas, ya que en ese punto es el mismo método `fibonacci` el que inicia las llamadas. Cada vez que se hace una llamada a `fibonacci`, se evalúan inmediatamente los dos casos base: `numero` igual a 0 o `numero` igual a 1 (línea 9). Si esta condición es verdadera, `fibonacci` simplemente devuelve `numero` ya que `fibonacci(0)` es 0, y `fibonacci(1)` es 1. Lo interesante es que, si `numero` es mayor que 1, el paso recursivo genera *dos* llamadas recursivas (línea 12), cada una de ellas para un problema ligeramente más pequeño que el de la llamada original a `fibonacci`.

La figura 15.7 muestra cómo el método `fibonacci` evalúa `fibonacci( 3 )`. Observe que en la parte inferior de la figura, nos quedamos con los valores 1, 0 y 1; los resultados de evaluar los casos base. Los primeros dos valores de retorno (de izquierda a derecha), 1 y 0, se devuelven como los valores para las llamadas `fibonacci( 1 )` y `fibonacci( 0 )`. La suma 1 más 0 se devuelve como el valor de `fibonacci( 2 )`. Esto se suma al resultado (1)

de la llamada a `fibonacci( 1 )`, para producir el valor 2. Después, este valor final se devuelve como el valor de `fibonacci( 3 )`.

La figura 15.7 genera ciertas preguntas interesantes, en cuanto al orden en el que los compiladores de Java evalúan los operandos de los operadores. Este orden es distinto del orden en el que se aplican los operadores a sus operandos; a saber, el orden que dictan las reglas de la precedencia de operadores. De la figura 15.7, parece ser que mientras se evalúa `fibonacci( 3 )`, se harán dos llamadas recursivas: `fibonacci( 2 )` y `fibonacci( 1 )`. Pero ¿en qué orden se harán estas llamadas? El lenguaje Java especifica que el orden de evaluación de los operandos es de izquierda a derecha. Por ende, la llamada a `fibonacci( 2 )` se realiza primero, y después la llamada a `fibonacci( 1 )`.

Hay que tener cuidado con los programas recursivos como el que utilizamos aquí para generar números de Fibonacci. Cada invocación del método `fibonacci` que no coincide con uno de los casos base (0 o 1) produce dos llamadas recursivas más al método `fibonacci`. Por lo tanto, este conjunto de llamadas recursivas se sale rápidamente de control. Para calcular el valor 20 de Fibonacci con el programa de la figura 15.5, se requieren 21,891 llamadas al método `fibonacci`; ¡para calcular el valor 30 de Fibonacci se requieren 2,692,537 llamadas! A medida que trate de calcular valores más grandes de Fibonacci, observará que cada número de Fibonacci consecutivo que calcule con la aplicación requiere un aumento considerable en tiempo de cálculo y en el número de llamadas al método `fibonacci`. Por ejemplo, el valor 31 de Fibonacci requiere 4,356,617 llamadas, y ¡el valor 32 de Fibonacci requiere 7,049,155 llamadas! Como puede ver, el número de llamadas al método `fibonacci` se incrementa con rapidez; 1,664,080 llamadas adicionales entre los valores 30 y 31 de Fibonacci, y ¡2,692,538 llamadas adicionales entre los valores 31 y 32 de Fibonacci! La diferencia en el número de llamadas realizadas entre los valores 31 y 32 de Fibonacci es de más de 1.5 veces la diferencia en el número de llamadas para los valores entre 30 y 31 de Fibonacci. Los problemas de esta naturaleza pueden humillar incluso hasta a las computadoras más poderosas del mundo. [*Nota:* en el campo de la teoría de la complejidad, los científicos de computadoras estudian qué tanto tienen que trabajar los algoritmos para completar sus tareas. Las cuestiones relacionadas con la complejidad se discuten con detalle en un curso del plan de estudios de ciencias computacionales de nivel superior, al que generalmente se le llama "Algoritmos". En el capítulo 16, Busqueda y ordenamiento, presentamos varias cuestiones acerca de la complejidad]. En los ejercicios le pediremos que mejore el programa de Fibonacci de la figura 15.5, de tal forma que calcule el monto aproximado de tiempo requerido para realizar el cálculo. Para este fin, llamará al método `static` de `System` llamado `currentTimeMillis`, el cual no recibe argumentos y devuelve el tiempo actual de la computadora en milisegundos.

Tip de rendimiento 15.1

Evite los programas recursivos al estilo de Fibonacci, ya que producen una "explosión" exponencial de llamadas a métodos.

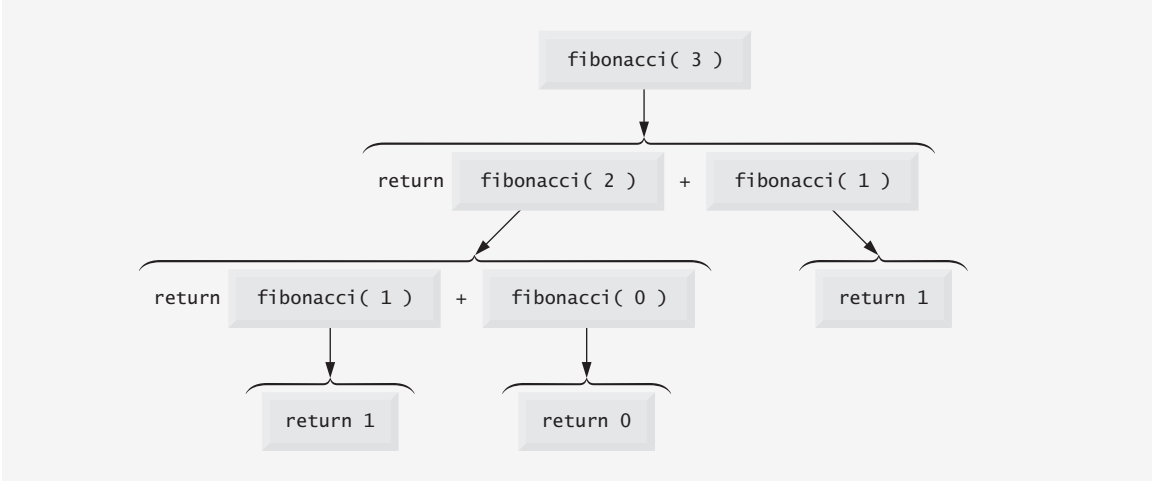


Figura 15.7 | Conjunto de llamadas recursivas para `fibonacci ( 3 )`.

15.5 La recursividad y la pila de llamadas a métodos

En el capítulo 6 se presentó la estructura de datos tipo pila, para comprender cómo Java realiza las llamadas a los métodos. Hablamos sobre la pila de llamadas a métodos (también conocida como la pila de ejecución del programa) y los registros de activación. En esta sección, utilizaremos estos conceptos para demostrar la forma en que la pila de ejecución del programa maneja las llamadas a los métodos recursivos.

Para empezar, regresemos al ejemplo de Fibonacci; en específico, la llamada al método `fibonacci` con el valor `3`, como en la figura 15.7. Para mostrar el orden en el que se colocan los registros de activación de las llamadas a los métodos en la pila, hemos clasificado las llamadas a los métodos con letras en la figura 15.8.

Cuando se hace la primera llamada al método (`A`), un registro de activación se mete en la pila de ejecución del programa, que contiene el valor de la variable local `numero` (`3` en este caso). La pila de ejecución del programa, que incluye el registro de activación para la llamada `A` al método, se ilustra en la parte (a) de la figura 15.9. [*Nota:* aquí utilizamos una pila simplificada. En una computadora real, la pila de ejecución del programa y sus registros de activación serían más complejos que en la figura 15.9, ya que contienen información como la ubicación a la que va a regresar la llamada al método cuando haya terminado de ejecutarse].

Dentro de la llamada al método `A` se realizan las llamadas `B` y `E`. La llamada original al método no se ha completado, por lo que su registro de activación permanece en la pila. La primera llamada al método en realizarse desde el interior de `A` es la llamada `B` al método, por lo que se mete el registro de activación para la llamada `B` al método en la pila, encima del registro de activación para la llamada `A` al método. La llamada `B` al método debe ejecutarse y completarse antes de realizar la llamada `E`. Dentro de la llamada `B` al método, se harán las llamadas `C`

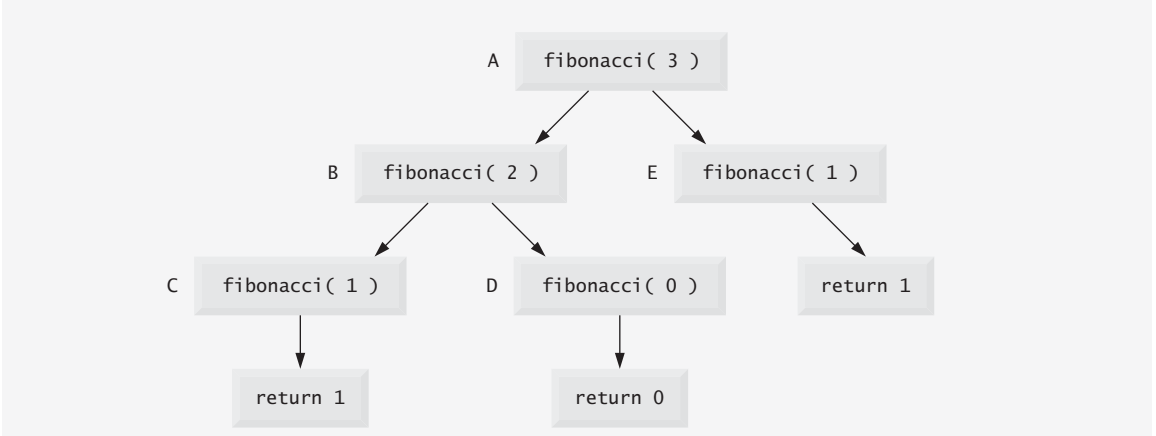


Figura 15.8 | Llamadas al método realizadas dentro de la llamada `fibonacci( 3 )`.

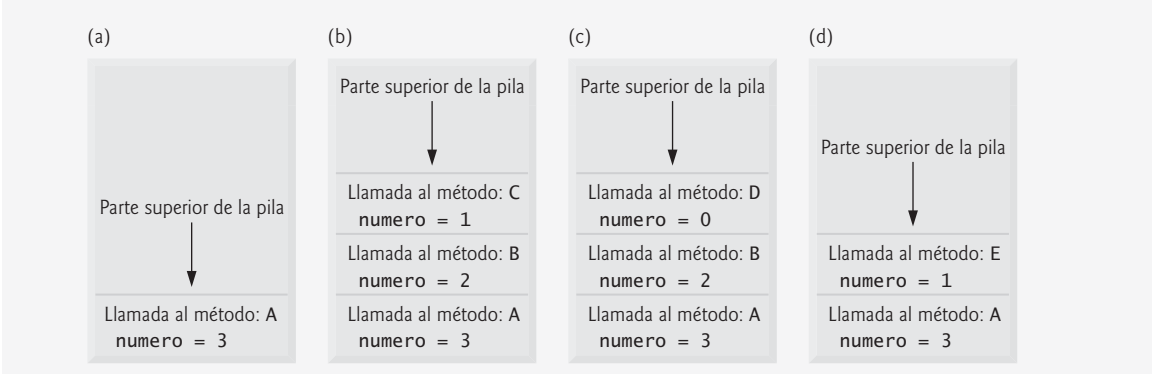


Figura 15.9 | Llamadas al método en la pila de ejecución del programa.

y D al método. La llamada C se realiza primero, y su registro de activación se mete en la pila [parte (b) de la figura 15.9]. La llamada B al método todavía no ha terminado, y su registro de activación sigue en la pila de llamadas a métodos. Cuando se ejecuta la llamada C, no realiza ninguna otra llamada al método, sino que simplemente devuelve el valor 1. Cuando este método regresa, su registro de activación se saca de la parte superior de la pila. La llamada al método en la parte superior de la pila es ahora B, que continúa ejecutándose y realiza la llamada D al método. El registro de activación para la llamada D se mete en la pila [parte (c) de la figura 15.9]. La llamada D al método se completa sin realizar ninguna otra llamada, y devuelve el valor 0. Después, se saca el registro de activación para esta llamada de la pila. Ahora, ambas llamadas al método que se realizaron desde el interior de la llamada B al método han regresado. La llamada B continúa ejecutándose, y devuelve el valor 1. La llamada B al método se completa y su registro de activación se saca de la pila. En este punto, el registro de activación para la llamada A al método se encuentra en la parte superior de la pila, y el método continúa su ejecución. Este método realiza la llamada E al método, cuyo registro de activación se mete ahora en la pila [parte (d) de la figura 15.9]. La llamada E al método se completa y devuelve el valor 1. El registro de activación para esta llamada al método se saca de la pila, y una vez más la llamada A al método continúa su ejecución. En este punto, la llamada A no realizará ninguna otra llamada al método y puede terminar su ejecución, para lo cual devuelve el valor 2 al método que llamó a A (`fibonacci(3) = 2`). El registro de activación de A se saca de la pila. Observe que el método en ejecución es siempre el que tiene su registro de activación en la parte superior de la pila, y el registro de activación para ese método contiene los valores de sus variables locales.

15.6 Comparación entre recursividad e iteración

En las secciones anteriores estudiamos los métodos `factorial` y `fibonacci`, que pueden implementarse fácilmente, ya sea en forma recursiva o iterativa. En esta sección compararemos los dos métodos, y veremos por qué le convendría al programador elegir un método en vez del otro, en una situación específica.

Tanto la iteración como la recursividad se basan en una instrucción de control: la iteración utiliza una instrucción de repetición (`for`, `while` o `do...while`), mientras que la recursividad utiliza una instrucción de selección (`if`, `if...else` o `switch`). Tanto la iteración como la recursividad implican la repetición: la iteración utiliza de manera explícita una instrucción de repetición, mientras que la recursividad logra la repetición a través de llamadas repetidas al método. La iteración y la recursividad implican una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo, mientras que la recursividad termina cuando se llega a un caso base. La iteración con repetición controlada por contador y la recursividad llegan en forma gradual a la terminación: la iteración sigue modificando un contador, hasta que éste asume un valor que hace que falle la condición de continuación de ciclo, mientras que la recursividad sigue produciendo versiones cada vez más pequeñas del problema original, hasta que se llega a un caso base. Tanto la iteración como la recursividad pueden ocurrir infinitamente: un ciclo infinito ocurre con la iteración si la prueba de continuación de ciclo nunca se vuelve falsa, mientas que la recursividad infinita ocurre si el paso recursivo no reduce el problema cada vez, de forma tal que llegue a converger en el caso base, o si el caso base no se evalúa.

Para ilustrar las diferencias entre la iteración y la recursividad, examinaremos una solución iterativa para el problema del factorial (figuras 15.10 y 15.11). Observe que se utiliza una instrucción de repetición (líneas 12 y 13 de la figura 15.10), en vez de la instrucción de selección de la solución recursiva (líneas 9 a 12 de la figura 15.3). Observe que ambas soluciones usan una prueba de terminación. En la solución recursiva, en la línea 9 se evalúa el caso base. En la solución iterativa, en la línea 12 se evalúa la condición de continuación de ciclo; si la prueba falla, el ciclo termina. Por último, observe que en vez de producir versiones cada vez más pequeñas del problema original, la solución iterativa utiliza un contador que se modifica hasta que la condición de continuación de ciclo se vuelve falsa.

```
1  // Fig. 15.10: CalculoFactorial.java
2  // Método factorial iterativo.
3
4  public class CalculoFactorial
5  {
6     // declaración recursiva del método factorial
```

Figura 15.10 | Solución de factorial iterativa. (Parte 1 de 2).

```
 7     public long factorial( long numero )
 8     {
 9        long resultado = 1;
10
11        // declaración iterativa del método factorial
12        for ( long i = numero; i >= 1; i-- )
13           resultado *= i;
14
15        return resultado;
16     } // fin del método factorial
17
18     // muestra los factoriales para los valores del 0 al 10
19     public void mostrarFactoriales()
20     {
21        // calcula los factoriales del 0 al 10
22        for ( int contador = 0; contador <= 10; contador++ )
23           System.out.printf( "%d! = %d\n", contador, factorial( contador ) );
24     } // fin del método mostrarFactoriales
25  } // fin de la clase CalculoFactorial
```

Figura 15.10 | Solución de factorial iterativa. (Parte 2 de 2).

```
 1  // Fig. 15.11: PruebaFactorial.java
 2  // Prueba del método factorial iterativo.
 3
 4  public class PruebaFactorial
 5  {
 6     // calcula los factoriales del 0 al 10
 7     public static void main( String args[] )
 8     {
 9        CalculoFactorial calculoFactorial = new CalculoFactorial();
10        calculoFactorial.mostrarFactoriales();
11     } // fin de main
12  } // fin de la clase PruebaFactorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Figura 15.11 | Prueba de la solución de factorial iterativa.

La recursividad tiene muchas desventajas. Invoca al mecanismo en forma repetida, y en consecuencia se produce una sobrecarga de las llamadas al método. Esta repetición puede ser perjudicial, en términos de tiempo del procesador y espacio de la memoria. Cada llamada recursiva crea otra copia del método (en realidad, sólo las variables del mismo, que se almacenan en el registro de activación); este conjunto de copias puede consumir una cantidad considerable de espacio en memoria. Como la iteración ocurre dentro de un método, se evitan las llamadas repetidas al método y la asignación adicional de memoria. Entonces, ¿por qué elegir la recursividad?

Observación de ingeniería de software 15.1

Cualquier problema que se pueda resolver mediante la recursividad, se puede resolver también mediante la iteración (sin recursividad). Por lo general, se prefiere un método recursivo a uno iterativo cuando el primero refleja con más naturalidad el problema, y se produce un programa más fácil de entender y de depurar. A menudo, puede implementarse un método recursivo con menos líneas de código. Otra razón por la que es preferible elegir un método recursivo es que uno iterativo podría no ser aparente.

Tip de rendimiento 15.2

Evite usar la recursividad en situaciones en las que se requiera un alto rendimiento. Las llamadas recursivas requieren tiempo y consumen memoria adicional.

Error común de programación 15.2

Hacer que un método no recursivo se llame a sí mismo por accidente, ya sea en forma directa o indirecta a través de otro método, puede provocar recursividad infinita.

15.7 Las torres de Hanoi

En las secciones anteriores de este capítulo, estudiamos métodos que pueden implementarse con facilidad, tanto en forma recursiva como iterativa. En esta sección presentamos un problema cuya solución recursiva demuestra la elegancia de la recursividad, y cuya solución iterativa tal vez no sea tan aparente.

Las **torres de Hanoi** son uno de los problemas clásicos con los que todo científico computacional en ciernes tiene que lidiar. Cuenta la leyenda que en un templo del Lejano Oriente, los sacerdotes intentan mover una pila de discos dorados, de una aguja de diamante a otra (figura 15.12). La pila inicial tiene 64 discos insertados en una aguja y se ordenan de abajo hacia arriba, de mayor a menor tamaño. Los sacerdotes intentan mover la pila de una aguja a otra, con las restricciones de que sólo se puede mover un disco a la vez, y en ningún momento se puede colocar un disco más grande encima de uno más pequeño. Se cuenta con tres agujas, una de las cuales se utiliza para almacenar discos temporalmente. Se supone que el mundo acabará cuando los sacerdotes completen su tarea, por lo que hay pocos incentivos para que nosotros podamos facilitar sus esfuerzos.

Supongamos que los sacerdotes intentan mover los discos de la aguja 1 a la aguja 2. Deseamos desarrollar un algoritmo que imprima la secuencia precisa de transferencias de los discos de una aguja a otra.

Si tratamos de encontrar una solución iterativa, es probable que terminemos "atados" manejando los discos sin esperanza. En vez de ello, si atacamos este problema mediante la recursividad podemos producir rápidamente una solución. La acción de mover *n* discos puede verse en términos de mover sólo *n* – 1 discos (de ahí la recursividad) de la siguiente forma:

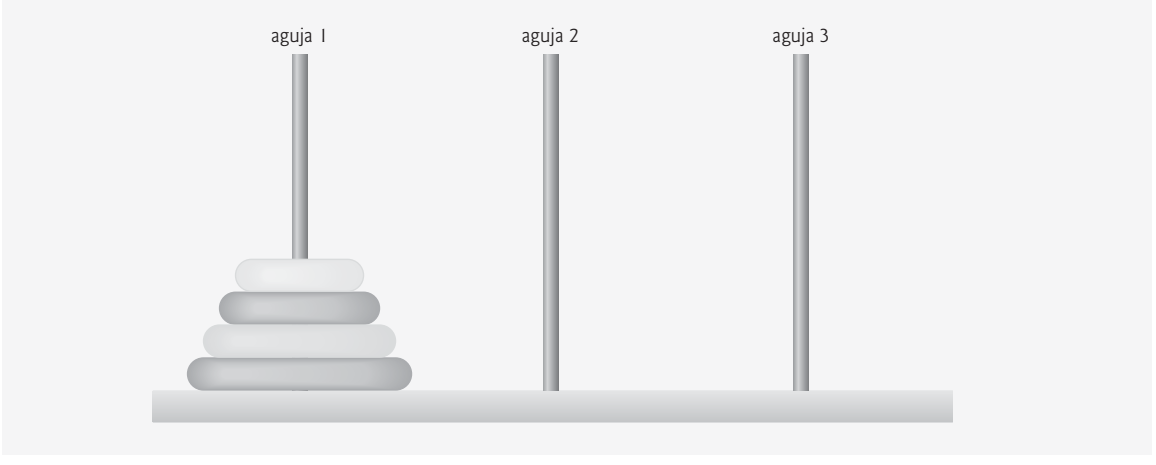


Figura 15.12 | Las torres de Hanoi para el caso con cuatro discos.

1. Mover $n - 1$ discos de la aguja 1 a la aguja 2, usando la aguja 3 como un área de almacenamiento temporal.
2. Mover el último disco (el más grande) de la aguja 1 a la aguja 3.
3. Mover $n - 1$ discos de la aguja 2 a la aguja 3, usando la aguja 1 como área de almacenamiento temporal.

El proceso termina cuando la última tarea implica mover $n = 1$ disco (es decir, el caso base). Esta tarea se logra con sólo mover el disco, sin necesidad de un área de almacenamiento temporal.

El programa de las figuras 15.13 y 15.14 resuelve las torres de Hanoi. En el constructor (líneas 9 a 12) se inicializa el número de discos a mover (`numDiscos`). El método `resolverTorres` (líneas 15 a 34) resuelve el acertijo de las torres de Hanoi, dado el número total de discos (en este caso 3), la aguja inicial, la aguja final y la aguja de almacenamiento temporal como parámetros. El caso base (líneas 19 a 23) ocurre cuando sólo se necesita mover un disco de la aguja inicial a la aguja final. En el paso recursivo (líneas 27 a 33), la línea 27 mueve `discos - 1` discos de la primera aguja (`agujaOrigen`) a la aguja de almacenamiento temporal (`agujaTemp`). Cuando se han movido todos los discos a la aguja temporal excepto uno, en la línea 30 se mueve el disco más grande de la aguja inicial a la aguja de destino. En la línea 33 se termina el resto de los movimientos, llamando al método `resolverTorres` para mover `discos - 1` discos de manera recursiva, de la aguja temporal (`agujaTemp`) a la aguja de destino (`agujaDestino`), esta vez usando la primera aguja (`agujaOrigen`) como aguja temporal.

```
1  // Fig. 15.13: TorresDeHanoi.java
2  // Programa que resuelve el problema de las torres de Hanoi, y
3  // demuestra la recursividad.
4
5  public class TorresDeHanoi
6  {
7     private int numDiscos; // número de discos a mover
8
9     public TorresDeHanoi( int discos )
10    {
11       numDiscos = discos;
12    } // fin del constructor de TorresDeHanoi
13
14    // mueve discos de una torre a otra, de manera recursiva
15    public void resolverTorres( int discos, int agujaOrigen, int agujaDestino,
16       int agujaTemp )
17    {
18       // caso base -- sólo hay que mover un disco
19       if ( discos == 1 )
20       {
21          System.out.printf( "\n%d --> %d", agujaOrigen, agujaDestino );
22          return;
23       } // fin de if
24
25       // paso recursivo -- mueve (disco - 1) discos de agujaOrigen
26       // a agujaTemp usando agujaDestino
27       resolverTorres( discos - 1, agujaOrigen, agujaTemp, agujaDestino );
28
29       // mueve el último disco de agujaOrigen a agujaDestino
30       System.out.printf( "\n%d --> %d", agujaOrigen, agujaDestino );
31
32       // mueve ( discos - 1 ) discos de agujaTemp a agujaDestino
33       resolverTorres( discos - 1, agujaTemp, agujaDestino, agujaOrigen );
34    } // fin del método resolverTorres
35 } // fin de la clase TorresDeHanoi
```

Figura 15.13 | Solución de las torres de Hanoi, con un método recursivo.

La figura 15.14 prueba nuestra solución de las torres de Hanoi. La línea 12 crea un objeto torres de Hanoi, pasando como parámetro el número total de los discos que se deben mover de una aguja a otra. La línea 15 llama al método recursivo `resolverTorres`, el cual muestra, al apuntador de comando, los pasos a seguir.

```
1  // Fig. 15.14: PruebaTorresDeHanoi.java
2  // Prueba la solución al problema de las torres de Hanoi.
3
4  public class PruebaTorresDeHanoi
5  {
6     public static void main( String args[] )
7     {
8        int agujaInicial = 1; // se usa el valor 1 para indicar agujaInicial en la salida
9        int agujaFinal = 3;   // se usa el valor 3 para indicar agujaFinal en la salida
10       int agujaTemp = 2;    // se usa el valor 2 para indicar agujaTemp en la salida
11       int totalDiscos = 3;  // número de discos
12       TorresDeHanoi torresDeHanoi = new TorresDeHanoi( totalDiscos );
13
14       // llamada no recursiva inicial: mueve todos los discos.
15       torresDeHanoi.resolverTorres( totalDiscos, agujaInicial, agujaFinal, agujaTemp );
16    } // fin de main
17 } // fin de la clase PruebaTorresDeHanoi
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Figura 15.14 | Prueba de la solución de las torres de Hanoi.

15.8 Fractales

Un **fractal** es una figura geométrica que se puede generar a partir de un patrón que se repite en forma recursiva (figura 15.15). Para modificar la figura, se aplica el patrón a cada segmento de la figura original. En esta sección analizaremos unas cuantas aproximaciones. [*Nota:* nos referiremos a nuestras figuras geométricas como fractales, aun cuando son aproximaciones]. Aunque estas figuras se han estudiado desde antes del siglo 20, fue el matemático polaco Benoit Mandelbrot quien introdujo el término "fractal" en la década de 1970, junto con los detalles específicos acerca de cómo se crea un fractal, y la aplicación práctica de los fractales. La geometría fractal de Mandelbrot proporciona modelos matemáticos para muchas formas complejas que se encuentran en la naturaleza, como las montañas, nubes y litorales. Los fractales tienen muchos usos en las matemáticas y la ciencia. Pueden utilizarse para comprender mejor los sistemas o patrones que aparecen en la naturaleza (por ejemplo, los ecosistemas), en el cuerpo humano (por ejemplo, en los pliegues del cerebro) o en el universo (por ejemplo, los grupos de galaxias). No todos los fractales se asemejan a los objetos en la naturaleza. El dibujo de fractales se ha convertido en una forma de arte popular. Los fractales tienen una **propiedad auto-similar**: cuando se subdividen en partes, cada una se asemeja a una copia del todo, en un tamaño reducido. Muchos fractales producen una copia exacta del original cuando se amplía una porción de la imagen original; se dice que dicho fractal es **estrictamente auto-similar**. En la sección 15.11 se proporcionan vínculos para diversos sitios Web en los que hay discusiones y demostraciones de los fractales.

Como ejemplo, veamos un fractal popular, estrictamente auto-similar, conocido como la **Curva de Koch** (figura 15.15). Para formar este fractal, se elimina la tercera parte media de cada línea en el dibujo, y se sustituye con dos líneas que forman un punto, de tal forma que si permaneciera la tercera parte media de la línea original, se formaría un triángulo equilátero. A menudo, las fórmulas para crear fractales implican eliminar toda, o parte de,

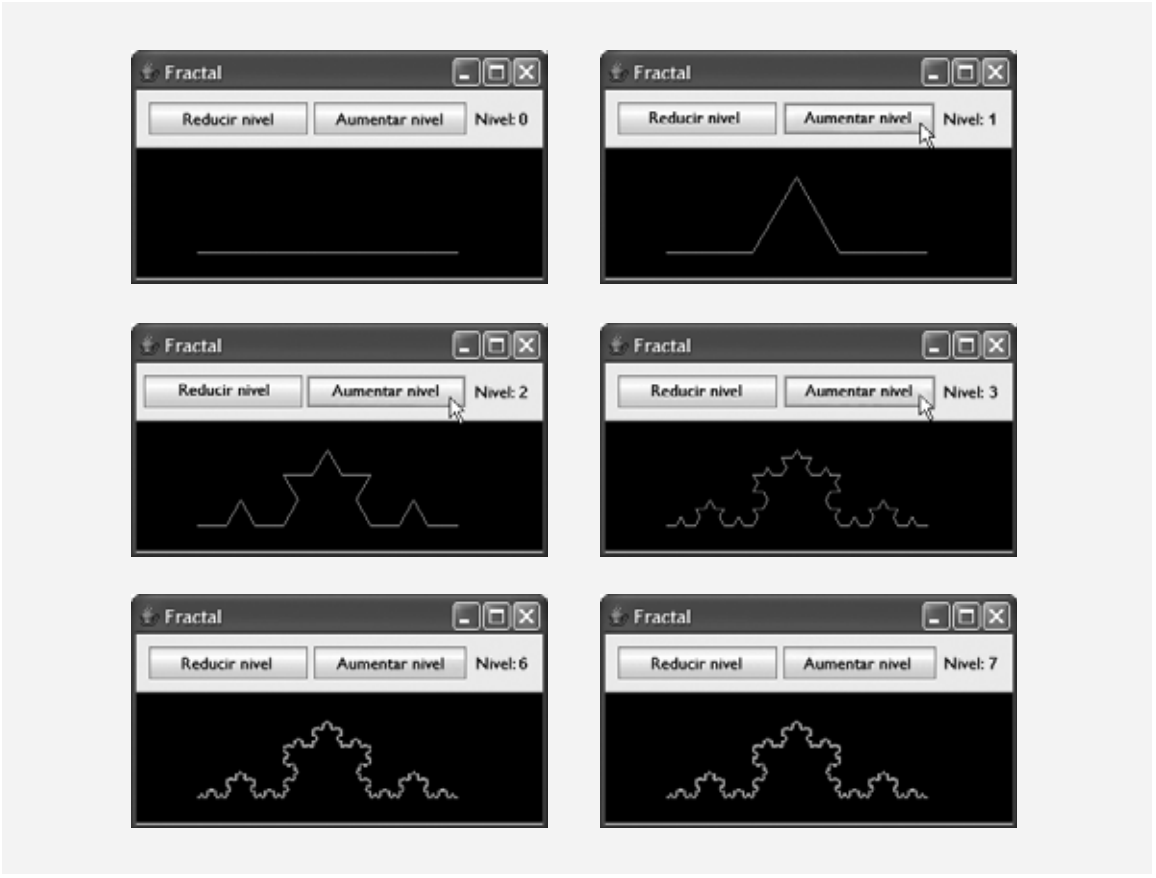


Figura 15.15 | Fractal `Curva de Koch`.

la imagen del fractal anterior. Este patrón ya se ha determinado para este fractal; en esta sección nos enfocaremos no sobre cómo determinar qué fórmulas se necesitan para un fractal específico, sino cómo utilizar esas fórmulas en una solución recursiva.

Empezamos con una línea recta [figura 15.15, parte (*a*)] y aplicamos el patrón, creando un triángulo a partir de la tercera parte media [figura 15.15, parte (*b*)]. Después aplicamos el patrón de nuevo a cada línea recta, lo cual produce la figura 15.15, parte (*c*). Cada vez que se aplica el patrón, decimos que el fractal está en un nuevo **nivel**, o **profundidad** (algunas veces se utiliza también el término **orden**). Los fractales pueden mostrarse en muchos niveles; por ejemplo, a un fractal de nivel 3 se le han aplicado tres iteraciones del patrón [figura 15.15, partes (*e* y *f*)]. Como éste es un fractal estrictamente auto-similar, cada porción del mismo contiene una copia exacta del fractal. Por ejemplo, en la parte (*f*) de la figura 15.15, hemos resaltado una porción del fractal con un cuadro color rojo punteado. Si se aumentara el tamaño de la imagen en este cuadro, se vería exactamente igual que el fractal completo de la parte (*f*).

Hay un fractal similar, llamado **Copo de nieve de Koch**, que es similar a la Curva de Koch, pero empieza con un triángulo en vez de una línea. Se aplica el mismo patrón a cada lado del triángulo, lo cual produce una imagen que se asemeja a un copo de nieve encerrado. Hemos optado por enfocarnos en la Curva de Koch por cuestión de simpleza. Para aprender más acerca de la Curva de Koch y del Copo de nieve de Koch, vea los vínculos de la sección 15.11.

Ahora demostraremos el uso de la recursividad para dibujar fractales, escribiendo un programa para crear un fractal estrictamente auto-similar. A este fractal lo llamaremos "fractal Lo", en honor de Sin Han Lo, un colega de Deitel & Associates que lo creó. En un momento dado, el fractal se asemejará a la mitad de una pluma (vea los resultados en la figura 15.22). El caso base, o nivel 0 del fractal, empieza como una línea entre dos puntos, `A` y `B` (figura 15.16). Para crear el siguiente nivel superior, buscamos el punto medio (`C`) de la línea. Para calcular

la ubicación del punto C, utilice la siguiente fórmula: [*Nota:* la x y la y a la izquierda de cada letra se refieren a las coordenadas *x* y *y* de ese punto, respectivamente. Por ejemplo, xA se refiere a la coordenada x del punto A, mientras que yC se refiere a la coordenada y del punto C. En nuestros diagramas denotamos el punto por su letra, seguida de dos números que representan las coordenadas *x* y *y*].

```
xC = (xA + xB) / 2;
yC = (yA + yB) / 2;
```

Para crear este fractal, también debemos buscar un punto D que se encuentre a la izquierda del segmento AC y que cree un triángulo recto isósceles ADC. Para calcular la ubicación del punto D, utilice las siguientes fórmulas:

```
xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
```

Ahora nos movemos del nivel 0 al nivel 1 de la siguiente manera: primero, se suman los puntos C y D (como en la figura 15.17). Después se elimina la línea original y se agregan los segmentos DA, DC y DB. El resto de las líneas se curvearán en un ángulo, haciendo que nuestro fractal se vea como una pluma. Para el siguiente nivel del fractal, este algoritmo se repite en cada una de las tres líneas en el nivel 1. Para cada línea, se aplican las fórmulas anteriores, en donde el punto anterior D se considera ahora como el punto A, mientras que el otro extremo de cada línea se considera como el punto B. La figura 15.18 contiene la línea del nivel 0 (ahora una línea punteada) y las tres líneas que se agregaron del nivel 1. Hemos cambiado el punto D para que sea el punto A, y los puntos originales A, C y B son B1, B2 y B3, respectivamente. Las fórmulas anteriores se han utilizado para buscar los nuevos puntos C y D en cada línea. Estos puntos también se enumeran del 1 al 3 para llevar la cuenta de cuál punto está asociado con cada línea. Por ejemplo, los puntos C1 y D1 representan a los puntos C y D asociados con la línea que se forma de los puntos A a B1. Para llegar al nivel 2, se eliminan las tres líneas de la figura 15.18 y se sustituyen con nuevas líneas de los puntos C y D que se acaban de agregar. La figura 15.19 muestra las nuevas líneas (las líneas del nivel 2 se muestran como líneas punteadas, para conveniencia del lector). La figura 15.20 muestra el nivel 2 sin las líneas punteadas del nivel 1. Una vez que se ha repetido este proceso varias veces, el fractal creado empezará a parecerse a la mitad de una pluma, como se muestra en los resultados de la figura 15.22. En breve presentaremos el código para esta aplicación.

La aplicación de la figura 15.21 define la interfaz de usuario para dibujar este fractal (que se muestra al final de la figura 15.22). La interfaz consiste de tres botones: uno para que el usuario modifique el color del fractal, otro para incrementar el nivel de recursividad y uno para reducir el nivel de recursividad. Un objeto JLabel lleva la cuenta del nivel actual de recursividad, que se modifica mediante una llamada al método establecerNivel, que veremos en breve. En las líneas 15 y 16 se especifican las constantes ANCHURA y ALTURA como 400 y 480 respectivamente, para indicar el tamaño del objeto JFrame. El color predeterminado para dibujar el fractal será azul (línea 18). El usuario activa un evento ActionEvent haciendo clic en el botón Color. El manejador de eventos para este botón se registra en las líneas 38 a 54. El método actionPerformed muestra un cuadro de diálogo JColorChoo-

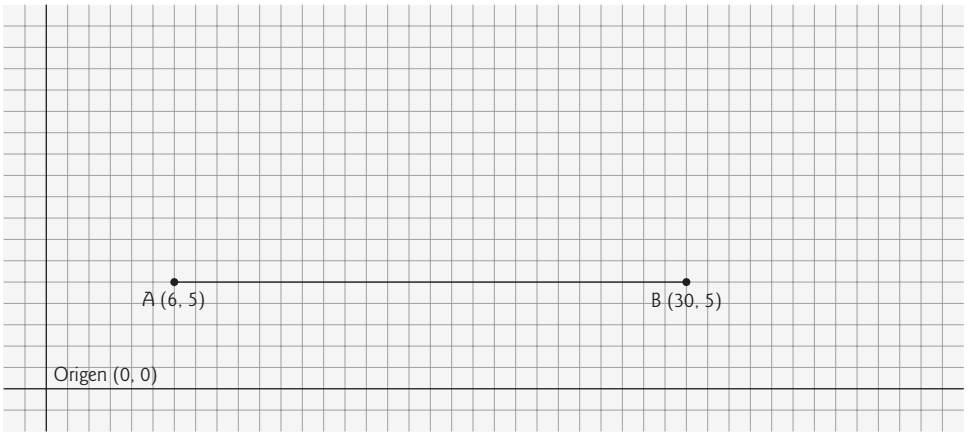


Figura 15.16 | El "fractal Lo" en el nivel 0.

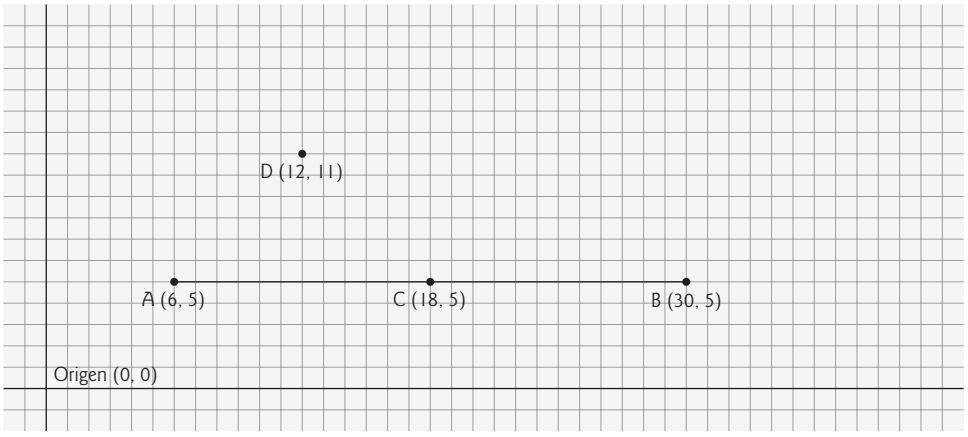


Figura 15.17 | Determinación de los puntos C y D para el nivel 1 del "fractal Lo".

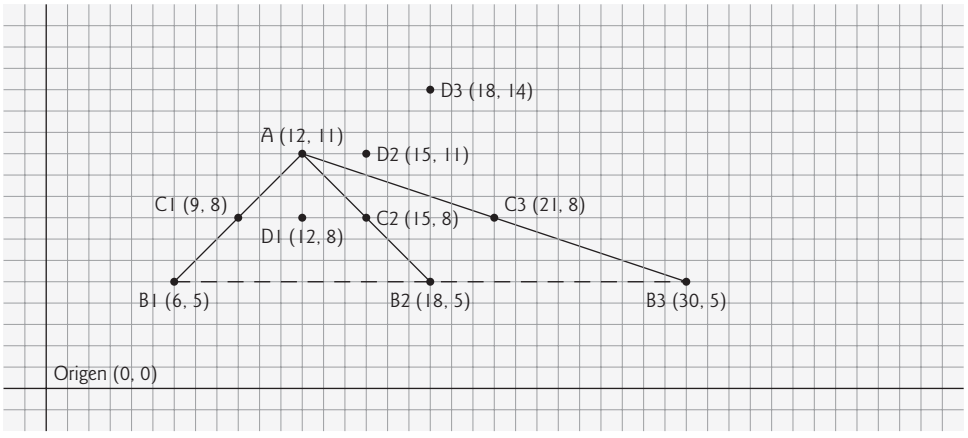


Figura 15.18 | El "fractal Lo" en el nivel 1, y se determinan los puntos C y D para el nivel 2. [*Nota:* se incluye el fractal en el nivel 0 como una línea punteada, para recordar en dónde se encontraba la línea en relación con el fractal actual].

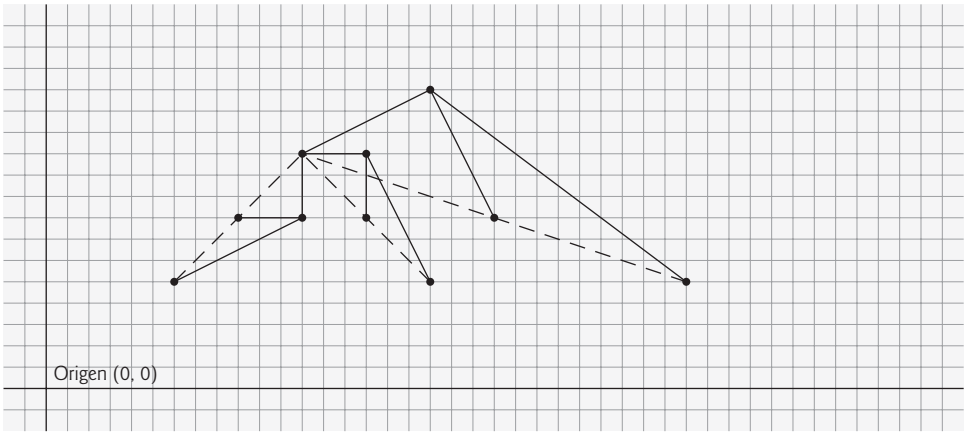


Figura 15.19 | El "fractal Lo" en el nivel 2, y se proporcionan las líneas punteadas del nivel 1.

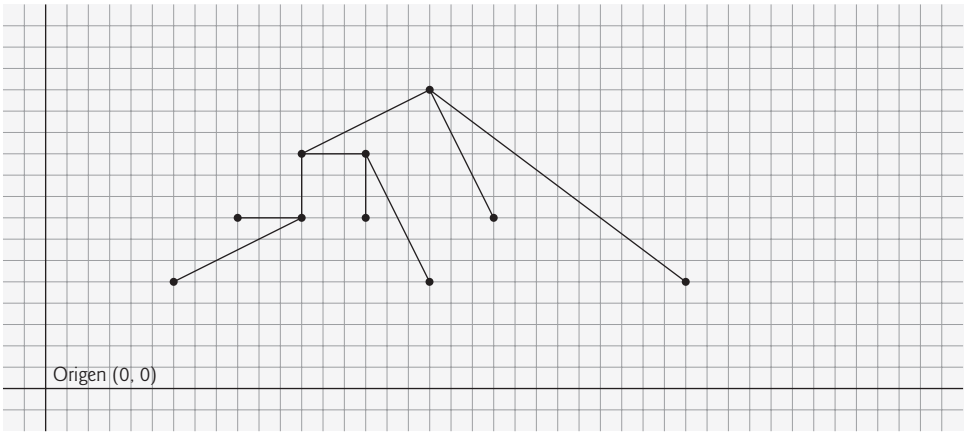


Figura 15.20 | El "fractal Lo" en el nivel 2.

ser. Este cuadro de diálogo devuelve el objeto **Color** seleccionado, o azul (si el usuario oprime **Cancelar** o cierra el cuadro de diálogo sin oprimir **Aceptar**). En la línea 51 se hace una llamada al método `establecerColor` en la clase `FractalJPanel` para actualizar el color.

El manejador de eventos para el botón **Reducir nivel** se registra en las líneas 60 a 78. En el método `actionPerformed`, en las líneas 66 y 67 obtienen el nivel actual de recursividad y lo reducen en 1. En la línea 70 se

```
1  // Fig. 15.21: Fractal.java
2  // Demuestra la interfaz de usuario para dibujar un fractal.
3  import java.awt.Color;
4  import java.awt.FlowLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JColorChooser;
12
13 public class Fractal extends JFrame
14 {
15    private final int ANCHURA = 400;     // define la anchura de la GUI
16    private final int ALTURA = 480;      // define la altura de la GUI
17    private final int NIVEL_MIN = 0, NIVEL_MAX = 15;
18    private Color color = Color.BLUE;
19
20    private JButton cambiarColorJButton, aumentarNivelJButton,
21       reducirNivelJButton;
22    private JLabel nivelJLabel;
23    private FractalJPanel espacioDibujo;
24    private JPanel principalJPanel, controlJPanel;
25
26    // establece la GUI
27    public Fractal()
28    {
29       super( "Fractal" );
30
```

Figura 15.21 | Demostración de la interfaz de usuario del fractal. (Parte 1 de 3).

```
31          // establece el panel de control
32          controlJPanel = new JPanel();
33          controlJPanel.setLayout( new FlowLayout() );
34
35          // establece el botón de color y registra el componente de escucha
36          cambiarColorJButton = new JButton( "Color" );
37          controlJPanel.add( cambiarColorJButton );
38          cambiarColorJButton.addActionListener(
39             new ActionListener() // clase interna anónima
40             {
41                // procesa el evento de cambiarColorJButton
42                public void actionPerformed( ActionEvent evento )
43                {
44                   color = JColorChooser.showDialog(
45                      Fractal.this, "Elija un color", color );
46
47                   // establece el color predeterminado, si no se devuelve un color
48                   if ( color == null )
49                      color = Color.BLUE;
50
51                   espacioDibujo.establecerColor( color );
52                } // fin del método actionPerformed
53             } // fin de la clase interna anónima
54          ); // fin de addActionListener
55
56          // establece botón para reducir nivel, para agregarlo al panel de control y
57          // registra el componente de escucha
58          reducirNivelJButton = new JButton( "Reducir nivel" );
59          controlJPanel.add( reducirNivelJButton );
60          reducirNivelJButton.addActionListener(
61             new ActionListener() // clase interna anónima
62             {
63                // procesa el evento de reducirNivelJButton
64                public void actionPerformed( ActionEvent evento )
65                {
66                   int nivel = espacioDibujo.obtenerNivel();
67                   nivel--; // reduce el nivel en uno
68
69                    // modifica el nivel si es posible
70                   if ( ( nivel >= NIVEL_MIN ) && ( nivel <= NIVEL_MAX ) )
71                   {
72                      nivelJLabel.setText( "Nivel: " + nivel );
73                      espacioDibujo.establecerNivel( nivel );
74                      repaint();
75                   } // fin de if
76                } // fin del método actionPerformed
77             } // fin de la clase interna anónima
78          ); // fin de addActionListener
79
80          // establece el botón para aumentar nivel, para agregarlo al panel de control
81          // y registra el componente de escucha
82          aumentarNivelJButton = new JButton( "Aumentar nivel" );
83          controlJPanel.add( aumentarNivelJButton );
84          aumentarNivelJButton.addActionListener(
85             new ActionListener() // clase interna anónima
86             {
87                // procesa el evento de aumentarNivelJButton
88                public void actionPerformed( ActionEvent evento )
```

Figura 15.21 | Demostración de la interfaz de usuario del fractal. (Parte 2 de 3).

```
89              {
90                 int nivel = espacioDibujo.obtenerNivel();
91                 nivel++; // aumenta el nivel en uno
92
93                 // modifica el nivel si es posible
94                 if ( ( nivel >= NIVEL_MIN ) && ( nivel <= NIVEL_MAX ) )
95                 {
96                    nivelJLabel.setText( "Nivel: " + nivel );
97                    espacioDibujo.establecerNivel( nivel );
98                    repaint();
99                 } // fin de if
100             } // fin del método actionPerformed
101          } // fin de la clase interna anónima
102       ); // fin de addActionListener
103
104       // establece nivelJLabel para agregarlo a controlJPanel
105       nivelJLabel = new JLabel( "Nivel: 0" );
106       controlJPanel.add( nivelJLabel );
107
108       espacioDibujo = new FractalJPanel( 0 );
109
110       // crea principalJPanel para que contenga a controlJPanel y espacioDibujo
111       principalJPanel = new JPanel();
112       principalJPanel.add( controlJPanel );
113       principalJPanel.add( espacioDibujo );
114
115       add( principalJPanel ); // agrega JPanel a JFrame
116
117       setSize( ANCHURA, ALTURA ); // establece el tamaño de JFrame
118       setVisible( true ); // muestra JFrame
119    } // fin del constructor de Fractal
120
121    public static void main( String args[] )
122    {
123       Fractal demo = new Fractal();
124       demo.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
125    } // fin de main
126 } // fin de la clase Fractal
```

Figura 15.21 | Demostración de la interfaz de usuario del fractal. (Parte 3 de 3).

realiza una verificación, para asegurar que el nivel sea mayor o igual que 0 (`NIVEL_MIN`); el fractal no está definido para cualquier nivel de recursividad menor que 0. El programa permite al usuario avanzar hacia cualquier nivel deseado, pero en cierto punto (nivel 10 y superior en este ejemplo) el despliegue del fractal se vuelve cada vez más lento, ya que hay muchos detalles que dibujar. En las líneas 72 a 74 se restablece la etiqueta del nivel para reflejar el cambio; se establece el nuevo nivel y se hace una llamada al método `repaint` para actualizar la imagen y mostrar el fractal correspondiente al nuevo nivel.

El objeto `JButton` **Aumentar nivel** funciona de la misma forma que el objeto `JButton` **Reducir nivel**, excepto que el nivel se incrementa en vez de reducirse para mostrar más detalles del fractal (líneas 90 y 91). Cuando se ejecuta la aplicación por primera vez, el nivel se establece en 0, en donde se muestra una línea azul entre dos puntos especificados en la clase `FractalJPanel`.

La clase `FractalJPanel` de la figura 15.22 especifica las medidas del objeto `JPanel` del dibujo como 400 por 400 (líneas 13 y 14). El constructor de `FractalJPanel` (líneas 18 a 24) recibe el nivel actual como parámetro y lo asigna a su variable de instancia `nivel`. La variable de instancia `color` se establece en el color azul predeterminado. En las líneas 22 y 23 se cambia el color de fondo del objeto `JPanel` para que sea blanco (para la visibilidad de los colores utilizados para dibujar el fractal), y se establecen las nuevas medidas del objeto `JPanel`, en donde se dibujará el fractal.

```
1  // Fig. 15.22: FractalJPanel.java
2  // FractalJPanel demuestra el dibujo recursivo de un fractal.
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import java.awt.Dimension;
6  import javax.swing.JPanel;
7
8  public class FractalJPanel extends JPanel
9  {
10    private Color color; // almacena el color utilizado para dibujar el fractal
11    private int nivel;   // almacena el nivel actual del fractal
12
13    private final int ANCHURA = 400;  // define la anchura de JPanel
14    private final int ALTURA = 400;   // define la altura de JPanel
15
16    // establece el nivel inicial del fractal al valor especificado
17    // y establece las especificaciones del JPanel
18    public FractalJPanel( int nivelActual )
19    {
20       color = Color.BLUE;  // inicializa el color de dibujo en azul
21       nivel = nivelActual; // establece el nivel inicial del fractal
22       setBackground( Color.WHITE );
23       setPreferredSize( new Dimension( ANCHURA, ALTURA ) );
24    } // fin del constructor de FractalJPanel
25
26    // dibuja el fractal en forma recursiva
27    public void dibujarFractal( int nivel, int xA, int yA, int xB,
28       int yB, Graphics g )
29    {
30       // caso base: dibuja una línea que conecta dos puntos dados
31       if ( nivel == 0 )
32          g.drawLine( xA, yA, xB, yB );
33       else // paso recursivo: determina los nuevos puntos, dibuja el siguiente nivel
34       {
35          // calcula punto medio entre (xA, yA) y (xB, yB)
36          int xC = ( xA + xB ) / 2;
37          int yC = ( yA + yB ) / 2;
38
39          // calcula el cuarto punto (xD, yD) que forma un
40          // triángulo recto isósceles entre (xA, yA) y (xC, yC)
41          // en donde el ángulo recto está en (xD, yD)
42          int xD = xA + ( xC - xA ) / 2 - ( yC - yA ) / 2;
43          int yD = yA + ( yC - yA ) / 2 + ( xC - xA ) / 2;
44
45          // dibuja el Fractal en forma recursiva
46          dibujarFractal( nivel - 1, xD, yD, xA, yA, g );
47          dibujarFractal( nivel - 1, xD, yD, xC, yC, g );
48          dibujarFractal( nivel - 1, xD, yD, xB, yB, g );
49       } // fin de else
50    } // fin del método dibujarFractal
51
52    // inicia el dibujo del fractal
53    public void paintComponent( Graphics g )
54    {
55       super.paintComponent( g );
56
57       // dibuja el patrón del fractal
58       g.setColor( color );
59       dibujarFractal( nivel, 100, 90, 290, 200, g );
```

Figura 15.22 | Dibujo del "fractal Lo" mediante el uso de la recursividad. (Parte 1 de 3).

```
60      } // fin del método paintComponent
61
62      // establece el color de dibujo a c
63      public void establecerColor( Color c )
64      {
65         color = c;
66      } // fin del método setColor
67
68      // establece el nuevo nivel de recursividad
69      public void establecerNivel( int nivelActual )
70      {
71         nivel = nivelActual;
72      } // fin del método setLevel
73
74      // devuelve el nivel de recursividad
75      public int obtenerNivel()
76      {
77         return nivel;
78      } // fin del método getLevel
79   } // fin de la clase FractalJPanel
```

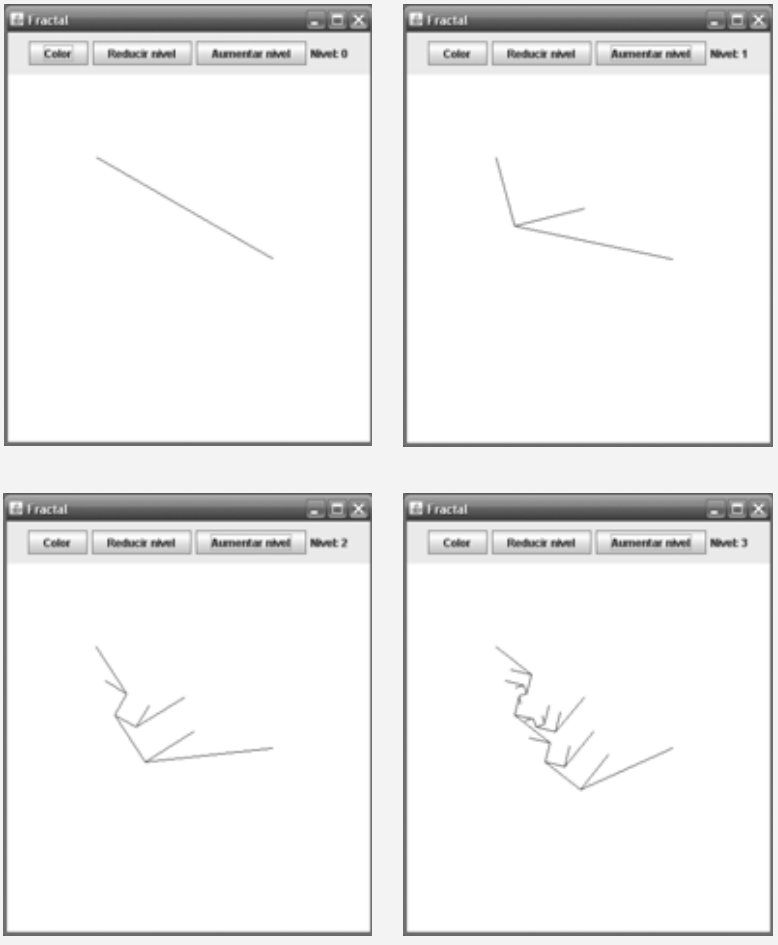


Figura 15.22 | Dibujo del "fractal Lo" mediante el uso de la recursividad. (Parte 2 de 3).

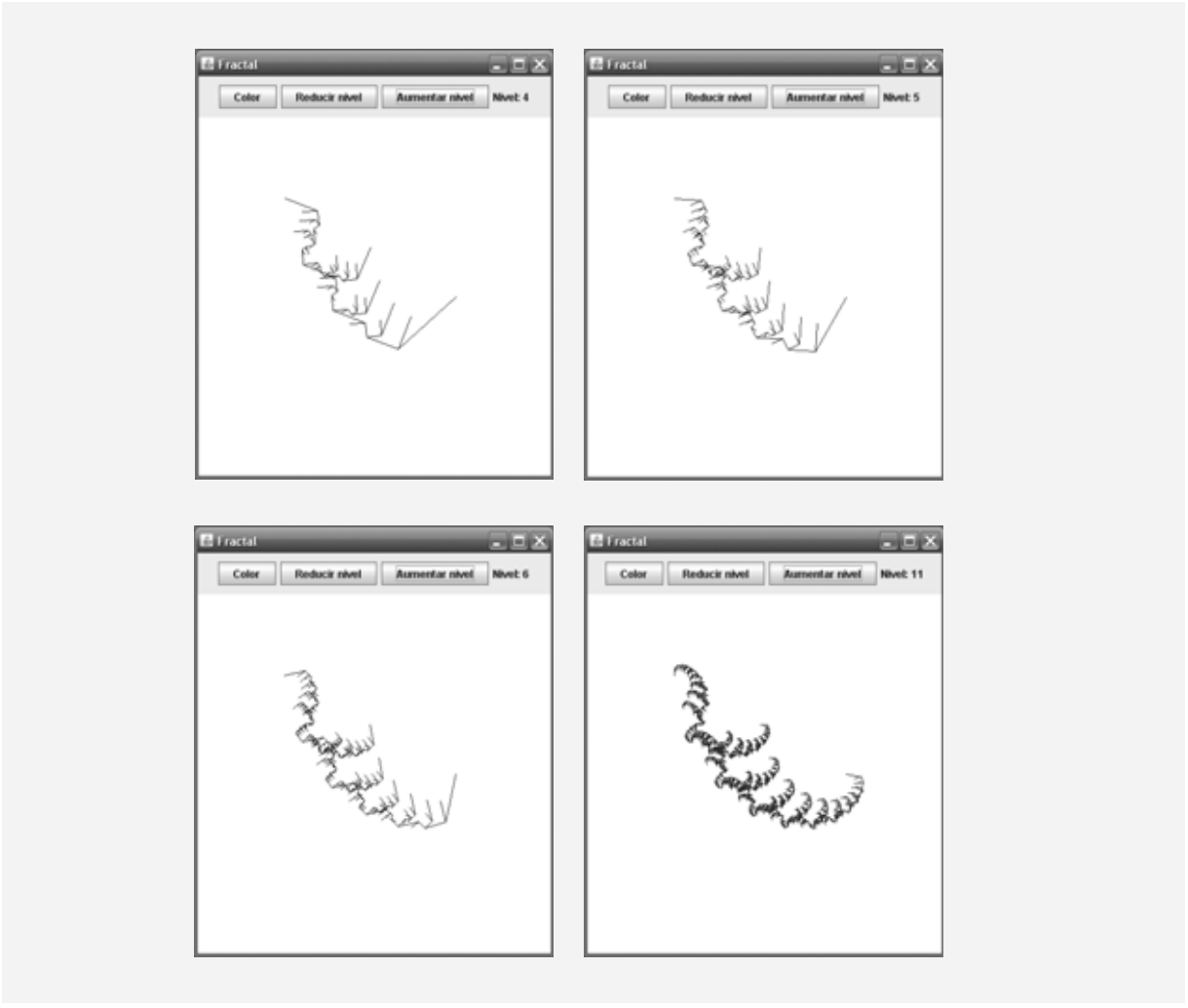


Figura 15.22 | Dibujo del "fractal Lo" mediante el uso de la recursividad. (Parte 3 de 3).

En las líneas 27 a 50 se define el método recursivo que crea el fractal. Este método recibe seis parámetros: el nivel, cuatro enteros que especifican las coordenadas *x* y *y* de dos puntos, y el objeto `g` de `Graphics`. El caso base para este método (línea 31) ocurre cuando `nivel` es igual a 0, en cuyo momento se dibujará una línea entre los dos puntos que se proporcionan como parámetros. En las líneas 36 a 43 se calcula (`xC`, `yC`), el punto medio entre (`xA`, `yA`) y (`xB`, `yB`), y (`xD`, `yD`), el punto que crea un triángulo isósceles recto con (`xA`, `yA`) y (`xC`, `yC`). En las líneas 46 a 48 se realizan tres llamadas recursivas en tres conjuntos distintos de puntos.

En el método `paintComponent`, en la línea 59 se realiza la primera llamada al método `dibujarFractal` para empezar el dibujo. Esta llamada al método no es recursiva, pero todas las llamadas subsiguientes a `dibujarFractal` que se realicen desde el cuerpo de `dibujarFractal` sí lo son. Como las líneas no se dibujarán sino hasta que se llegue al caso base, la distancia entre dos puntos se reduce en cada llamada recursiva. A medida que aumenta el nivel de recursividad, el fractal se vuelve más uniforme y detallado. La figura de este fractal se estabiliza a medida que el nivel se acerca a 11. Los fractales se estabilizarán en distintos niveles, con base en la figura y el tamaño del fractal.

En la figura 15.22 se muestra el desarrollo del fractal, de los niveles 0 al 6. La última imagen muestra la figura que define el fractal en el nivel 11. Si nos enfocamos en uno de los brazos de este fractal, será idéntico a la imagen completa. Esta propiedad define al fractal como estrictamente auto-similar. En la sección 15.11 podrá consultar más recursos acerca de los fractales.

15.9 "Vuelta atrás" recursiva (backtracking)

Todos nuestros métodos recursivos tienen una arquitectura similar; si se llega al caso base, devuelven un resultado; si no, hacen una o más llamadas recursivas. En esta sección exploraremos un método recursivo más completo, que busca una ruta a través de un laberinto, y devuelve verdadero si hay una posible solución al laberinto. La solución implica recorrer el laberinto un paso a la vez, en donde los movimientos pueden ser hacia abajo, a la derecha, hacia arriba o a la izquierda (no se permiten movimientos diagonales). De la posición actual en el laberinto (empezando con el punto de entrada), se realizan los siguientes pasos: se elije una dirección, se realiza el movimiento en esa dirección y se hace una llamada recursiva para resolver el resto del laberinto desde la nueva ubicación. Cuando se llega a un punto sin salida (es decir, no podemos avanzar más pasos sin pegar en la pared), retrocedemos a la ubicación anterior y tratamos de avanzar en otra dirección. Si no puede elegirse otra dirección, retrocedemos de nuevo. Este proceso continúa hasta que encontramos un punto en el laberinto en donde puede realizarse un movimiento en otra dirección. Una vez que se encuentra dicha ubicación, avanzamos en la nueva dirección y continuamos con otra llamada recursiva para resolver el resto del laberinto.

Para retroceder a la ubicación anterior en el laberinto, nuestro método recursivo simplemente devuelve falso, avanzando hacia arriba en la cadena de llamadas a métodos, hasta la llamada recursiva anterior (que hace referencia a la ubicación anterior en el laberinto). A este proceso de utilizar la recursividad para regresar a un punto de decisión anterior se le conoce como **"vuelta atrás" recursiva**. Si un conjunto de llamadas recursivas no resulta en una solución para el problema, el programa retrocede hasta el punto de decisión anterior y toma una decisión distinta, lo que a menudo produce otro conjunto de llamadas recursivas. En este ejemplo, el punto de decisión anterior es la ubicación anterior en el laberinto, y la decisión a realizar es la dirección que debe tomar el siguiente movimiento. Una dirección ha conducido a un punto sin salida, por lo que la búsqueda continúa con una dirección diferente. A diferencia de nuestros demás programas recursivos, que llegaron al caso base y luego regresaron a través de toda la cadena de llamadas a métodos, hasta la llamada al método original, la solución de "vuelta atrás" recursiva para el problema del laberinto utiliza la recursividad para regresar sólo una parte a través de la cadena de llamadas a métodos, y después probar una dirección diferente. Si la vuelta atrás llega a la ubicación de entrada del laberinto y se han recorrido todas las direcciones, entonces el laberinto no tiene solución.

En los ejercicios del capítulo le pediremos que implemente soluciones de "vuelta atrás" recursivas para el problema del laberinto (ejercicios 15.20, 15.21 y 15.22) y para el problema de las Ocho Reinas (ejercicio 15.15), el cual trata de encontrar la manera de colocar ocho reinas en un tablero de ajedrez vacío, de forma que ninguna reina esté "atacando" a otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal). En la sección 15.11 podrá consultar vínculos hacia más información sobre la "vuelta atrás" recursiva.

15.10 Conclusión

En este capítulo aprendió a crear métodos recursivos; es decir, métodos que se llaman a sí mismos. Aprendió que los métodos recursivos generalmente dividen a un problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver (el caso base) y una pieza que el método no sabe cómo resolver (el paso recursivo). El paso recursivo es una versión ligeramente más pequeña del problema original, y se lleva a cabo mediante una llamada a un método recursivo. Vio algunos ejemplos populares de recursividad, incluyendo el cálculo de factoriales y la producción de valores en la serie de Fibonacci. Después aprendió cómo funciona la recursividad "detrás de las cámaras", incluyendo el orden en el que se meten o se sacan las llamadas a métodos recursivos de la pila de ejecución del programa. Después comparó los métodos recursivo e iterativo (no recursivo). Aprendió a resolver un problema más complejo mediante la recursividad: mostrar fractales. El capítulo concluyó con una introducción a la "vuelta atrás" recursiva, una técnica para resolver problemas que implica retroceder a través de llamadas recursivas para probar distintas soluciones posibles. En el siguiente capítulo, aprenderá diversas técnicas para ordenar listas de datos y buscar un elemento en una lista de datos, y bajo qué circunstancias debe utilizarse cada técnica de búsqueda y ordenamiento.

15.11 Recursos en Internet y Web

Conceptos de recursividad

`en.wikipedia.org/wiki/Recursion`

Artículo de Wikipedia, que proporciona los fundamentos de la recursividad y varios recursos para los estudiantes.

`es.wikipedia.org/wiki/Recursividad`
El mismo artículo anterior de Wikipedia, en español.
`www.cafeaulait.org/javatutorial.html`
Proporciona una breve introducción a la recursividad en Java, y también cubre otros temas relacionados con Java.

Pilas

`www.cs.auc.dk/~normark/eciu-recursion/html/recit-slide-implerec.html`
Proporciona diapositivas acerca de la implementación de la recursividad mediante el uso de pilas.
`faculty.juniata.edu/kruse/cs2java/recurimpl.htm`
Proporciona un diagrama de la pila de ejecución del programa y describe la forma en que funciona la pila.

Fractales

`math.rice.edu/~lanius/frac/`
Proporciona ejemplos de otros fractales, como el Copo de nieve de Koch, el Triángulo de Sierpinski y los fractales de Parque Jurásico.
`www.lifesmith.com/`
Proporciona cientos de imágenes de fractales coloridas, junto con una explicación detallada acerca de los conjuntos de Mandelbrot y Julia, dos conjuntos comunes de fractales.
`www.jracademy.com/~jtucek/math/fractals.html`
Contiene dos películas AVI creadas al realizar acercamientos continuos en los fractales, conocidos como los conjuntos de ecuaciones de Mandelbrot y Julia.
`www.faqs.org/faqs/fractal-faq/`
Proporciona las respuestas a muchas preguntas acerca de los fractales.
`spanky.triumf.ca/www/fractint/fractint.html`
Contiene vínculos para descargar Fractint, un programa de freeware para generar fractales.
`www.42explore.com/fractal.htn`
Proporciona una lista de URLs en fractales y herramientas de software que crean fractales.
`www.arcytech.org/java/fractals/koch.shtml`
Proporciona una introducción detallada al fractal de la Curva de Koch y un applet que demuestra el fractal.
`library.thinkquest.org/26688/koch.html`
Muestra un applet de la Curva de Koch, y proporciona el código fuente.

"Vuelta atrás" recursiva

`www.cs.sfu.ca/CourseCentral/201/havens/notes/Lecture14.pdf`
Proporciona una breve introducción a la "vuelta atrás" recursiva, incluyendo un ejemplo acerca de la planeación de una ruta de viaje.
`math.hws.edu/xJava/PentominosSolver`
Proporciona un programa que utiliza la "vuelta atrás" recursiva para resolver un problema, conocido como el acertijo de Pentominós (que se describe en el sitio).

Resumen

Sección 15.1 Introducción

- Un método recursivo se llama a sí mismo en forma directa o indirecta a través de otro método.
- Cuando se llama a un método recursivo para resolver un problema, en realidad el método es capaz de resolver sólo el (los) caso(s) más simple(s), o caso(s) base. Si se llama con un caso base, el método devuelve un resultado.

Sección 15.2 Conceptos de recursividad

- Si se llama a un método recursivo con un problema más complejo que el caso base, por lo general, divide el problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver y otra pieza que no sabe cómo resolver.

- Para que la recursividad sea factible, la pieza que el método no sabe cómo resolver debe asemejarse al problema original, pero debe ser una versión ligeramente más simple o pequeña del mismo. Como este nuevo problema se parece al problema original, el método llama a una nueva copia de sí mismo para trabajar en el problema más pequeño; a esto se le conoce como paso recursivo.
- Para que la recursividad termine en un momento dado, cada vez que un método se llame a sí mismo con una versión más simple del problema original, la secuencia de problemas cada vez más pequeños debe converger en un caso base. Cuando el método reconoce el caso base, devuelve un resultado a la copia anterior del método.
- Una llamada recursiva puede ser una llamada a otro método, que a su vez realiza una llamada de vuelta al método original. Dicho proceso sigue provocando una llamada recursiva al método original. A esto se le conoce como llamada recursiva indirecta, o recursividad indirecta.

Sección 15.3 Ejemplo de uso de recursividad: factoriales

- La acción de omitir el caso base, o escribir el paso recursivo de manera incorrecta para que no converja en el caso base, puede ocasionar una recursividad infinita, con lo cual se agota la memoria en cierto punto. Este error es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva).

Sección 15.4 Ejemplo de uso de recursividad: serie de Fibonacci

- La serie de Fibonacci empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos anteriores.
- La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número al que se le denomina la proporción dorada, o media dorada.
- Algunas soluciones recursivas, como la de Fibonacci (que realiza dos llamadas por cada paso recursivo), producen una "explosión" de llamadas a métodos.

Sección 15.5 La recursividad y la pila de llamadas a métodos

- Una pila es una estructura de datos en la que sólo se pueden agregar o eliminar datos de la parte superior.
- Una pila es la analogía de un montón de platos. Cuando se coloca un plato en el montón, siempre se coloca en la parte superior (a esto se le conoce como meter el plato en la pila). De manera similar, cuando se quita un plato del montón, siempre se quita de la parte superior (a esto se le conoce como sacar el plato de la pila).
- Las pilas se conocen como estructuras de datos "último en entrar, primero en salir" (UEPS): el último elemento que se metió (insertó) en la pila es el primero que se saca (elimina) de ella.
- Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, cuando un programa llama a un método, el método que se llamó debe saber cómo regresar al que lo llamó, por lo que se mete la dirección de retorno del método que hizo la llamada en la pila de ejecución del programa (a la que algunas veces se le conoce como la pila de llamadas a métodos).
- La pila de ejecución del programa contiene la memoria para las variables locales en cada invocación de un método, durante la ejecución de un programa. Estos datos, que se almacenan como una parte de la pila de ejecución del programa, se conocen como el registro de activación o marco de pila de la llamada al método.
- Si hay más llamadas a métodos recursivas o anidadas de las que pueden almacenarse en la pila de ejecución del programa, se produce un error conocido como desbordamiento de pila.

Sección 15.6 Comparación entre recursividad e iteración

- Tanto la iteración como la recursividad se basan en una instrucción de control: la iteración utiliza una instrucción de repetición, la recursividad una instrucción de selección.
- Tanto la iteración como la recursividad implican la repetición: la iteración utiliza de manera explícita una instrucción de repetición, mientras que la recursividad logra la repetición a través de llamadas repetidas a un método.
- La iteración y la recursividad implican una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo, la recursividad cuando se reconoce un caso base.
- La iteración con repetición controlada por contador y la recursividad se acercan en forma gradual a la terminación: la iteración sigue modificando un contador, hasta que éste asume un valor que hace que falle la condición de continuación de ciclo, mientras que la recursividad sigue produciendo versiones cada vez más simples del problema original, hasta llegar al caso base.
- Tanto la iteración como la recursividad pueden ocurrir en forma infinita. Un ciclo infinito ocurre con la iteración si la prueba de continuación de ciclo nunca se vuelve falsa, mientras que la recursividad infinita ocurre si el paso recursivo no reduce el problema cada vez más, de una forma que converja en el caso base.
- La recursividad invoca el mecanismo en forma repetida, y en consecuencia a la sobrecarga producida por las llamadas al método.

- Cualquier problema que pueda resolverse en forma recursiva, se puede resolver también en forma iterativa.
- Por lo general se prefiere un método recursivo en vez de uno iterativo cuando el primero refleja el problema con más naturalidad, y produce un programa más fácil de comprender y de depurar.
- A menudo se puede implementar un método recursivo con pocas líneas de código, pero el método iterativo correspondiente podría requerir una gran cantidad de código. Otra razón por la que es más conveniente elegir una solución recursiva es que una solución iterativa podría no ser aparente.

Sección 15.8 Fractales

- Un fractal es una figura geométrica que se genera a partir de un patrón que se repite en forma recursiva, un número infinito de veces.
- Los fractales tienen una propiedad de auto-similitud: las subpartes son copias de tamaño reducido de toda la pieza.

Sección 15.9 "Vuelta atrás" recursiva (backtracking)

- Al uso de la recursividad para regresar a un punto de decisión anterior se le conoce como "vuelta atrás" recursiva. Si un conjunto de llamadas recursivas no produce como resultado una solución al problema, el programa retrocede hasta el punto de decisión anterior y toma una decisión distinta, lo cual a menudo produce otro conjunto de llamadas recursivas.

Terminología

caso base
converger en un caso base
Copo de nieve de Koch, fractal
Curva de Koch, fractal
desbordamiento de pila
evaluación recursiva
factorial
Fibonacci, serie de
Fractal
fractal auto-similar
fractal estrictamente auto-similar
llamada recursiva
marco de pila
media dorada
método recursivo
nivel de un fractal
nivel del fractal
Ocho Reinas, problema
orden del fractal
palíndromo
paso recursivo
pila
pila de ejecución del programa
pila de llamadas a métodos
profundidad del fractal
proporción dorada
prueba de terminación
recorrido del laberinto, problema
recursividad exhaustiva
recursividad indirecta
recursividad infinita
registro de activación
sobrecarga de ejecución
teoría de complejidad
torres de Hanoi, problema
último en entrar, primero en salir (UEPS), estructuras de datos
"vuelta atrás"
"vuelta atrás" recursiva

Ejercicios de autoevaluación

15.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Un método que se llama a sí mismo en forma indirecta no es un ejemplo de recursividad.
b) La recursividad puede ser eficiente en la computación, debido a la reducción en el uso del espacio en memoria.
c) Cuando se llama a un método recursivo para resolver un problema, en realidad es capaz de resolver sólo el (los) caso(s) más simple(s), o caso(s) base.
d) Para que la recursividad sea factible, el paso recursivo en una solución recursiva debe asemejarse al problema original, pero debe ser una versión ligeramente más grande del mismo.

15.2 Para terminar la recursividad, se requiere un(a) _____.
a) paso recursivo
b) instrucción `break`
c) tipo de valor de retorno `void`
d) caso base

15.3 La primera llamada para invocar a un método recursivo es _____.
a) no recursiva
b) recursiva
c) el paso recursivo
d) ninguna de las anteriores

15.4 Cada vez que se aplica el patrón de un fractal, se dice que el fractal está en un(a) nuevo(a) _____.
a) anchura
b) altura
c) nivel
d) volumen

15.5 La iteración y la recursividad implican un(a) _____.
a) instrucción de repetición
b) prueba de terminación
c) variable contador
d) ninguna de las anteriores

15.6 Complete los siguientes enunciados:
a) La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número al que se le conoce como _____ o _____.
b) Sólo pueden agregarse o eliminarse datos de la _____ de la pila.
c) Las pilas se conocen como estructuras de datos _____; el último elemento que se metió (insertó) en la pila es el primer elemento que se saca (elimina) de ella.
d) La pila de ejecución del programa contiene la memoria para las variables locales en cada invocación de un método, durante la ejecución de un programa. Estos datos, que se almacenan como una parte de la pila de ejecución del programa, se conocen como el _____ o el _____ de llamadas a métodos.
e) Si hay más llamadas a métodos recursivas o anidadas de las que puedan almacenarse en la pila de ejecución del programa, se produce un error conocido como _____.
f) Por lo general, la iteración utiliza una instrucción de repetición, mientras que la recursividad comúnmente utiliza una instrucción _____.
g) Los fractales tienen una propiedad llamada _____; cuando se subdividen en partes, cada una de ellas es una copia de tamaño reducido de la pieza completa.
h) Las _____ de una cadena son todas las cadenas distintas que pueden crearse al reordenar los caracteres de la cadena original.
i) La pila de ejecución del programa se conoce también como la pila _____.

Respuestas a los ejercicios de autoevaluación

15.1 a) Falso. Un método que se llama a sí mismo en forma indirecta es un ejemplo de recursividad; en forma más específica, es un ejemplo de recursividad indirecta. b) Falso. La recursividad puede ser ineficiente en la computación debido a las múltiples llamadas a un método y el uso del espacio de memoria. c) Verdadero. d) Falso. Para que la recursividad sea factible, el paso recursivo en una solución recursiva debe asemejarse al problema original, pero debe ser una versión ligeramente *más pequeña* del mismo.

15.2 d

15.3 a

15.4 c

15.5 b

15.6 a) proporción dorada, media dorada. b) parte superior. c) último en entrar, primero en salir (UEPS). d) registro de activación, marco de pila. e) desbordamiento de pila. f) de selección. g) auto-similitud. h) permutaciones. i) de llamadas a métodos.

Ejercicios

15.7 ¿Qué hace el siguiente código?

```
1  public int misterio( int a, int b )
2  {
3     if ( b == 1 )
4        return a;
5     else
6        return a + misterio( a, b - 1 );
7  } // fin del método misterio
```

15.8 Busque el(los) error(es) en el siguiente método recursivo, y explique cómo corregirlo(s). Este método debe encontrar la suma de los valores de 0 a `n`.

```
1  public int suma( int n )
2  {
3     if ( n == 0 )
4        return 0;
5     else
6        return n + suma( n );
7  } // fin del método suma
```

15.9 *(Método `potencia` recursivo)* Escriba un método recursivo llamado `potencia( base, exponente )` que, cuando sea llamado, devuelva

$$base^{exponente}$$

Por ejemplo, `potencia( 3, 4 )` = 3 * 3 *3 * 3. Suponga que `exponente` es un entero mayor o igual que `1`. [*Sugerencia:* el paso recursivo debe utilizar la relación

$$base^{exponente} = base \cdot base^{exponente - 1}$$

y la condición de terminación ocurre cuando `exponente` es igual a `1`, ya que

$$base^{1} = base$$

Incorpore este método en un programa que permita al usuario introducir la `base` y el `exponente`].

15.10 *(Visualización de la recursividad)* Es interesante observar la recursividad "en acción". Modifique el método factorial de la figura 15.3 para imprimir su variable local y su parámetro de llamada recursiva. Para cada llamada recursiva, muestre los resultados en una línea separada y agregue un nivel de sangría. Haga su máximo esfuerzo por hacer que los resultados sean claros, interesantes y significativos. Su meta aquí es diseñar e implementar un formato de salida que facilite la comprensión de la recursividad. Tal vez desee agregar ciertas capacidades de visualización a otros ejemplos y ejercicios recursivos a lo largo de este libro.

15.11 *(Máximo común divisor)* El máximo común divisor de los enteros `x` y `y` es el entero más grande que se puede dividir entre `x` y `y` de manera uniforme. Escriba un método recursivo llamado `mcd`, que devuelva el máximo común divisor de `x` y `y`. El `mcd` de `x` y `y` se define, mediante la recursividad, de la siguiente manera: si `y` es igual a `0`, entonces `mcd( x, y )` es `x`; en caso contrario, `mcd( x, y )` es `mcd( y, x % y )`, en donde `%` es el operador residuo. Use este método para sustituir el que escribió en la aplicación del ejercicio 6.27.

15.12 ¿Qué hace el siguiente programa?

```
1  // Ejercicio 15.12 Solución: ClaseMisteriosa.java
2
3  public class ClaseMisteriosa
4  {
5     public int misterio( int arreglo2[], int tamanio )
6     {
7        if ( tamanio == 1 )
8           return arreglo2[ 0 ];
```

```
         else
            return arreglo2[ tamanio - 1 ] + misterio( arreglo2, tamanio - 1 );
      } // fin del método misterio
   } // fin de la clase ClaseMisteriosa
```

```
// Ejercicio 15.12 Solución: PruebaMisteriosa.java

public class PruebaMisteriosa
{
   public static void main( String args[] )
   {
      ClaseMisteriosa objetoMisterioso = new ClaseMisteriosa();

      int arreglo[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      int resultado = objetoMisterioso.misterio( arreglo, arreglo.length );

      System.out.printf( "El resultado es: %d\n", resultado );
   } // fin del método main
} // fin de la clase PruebaMisteriosa
```

15.13 ¿Qué hace el siguiente programa?

```
// Ejercicio 15.13 Solución: UnaClase.java

public class UnaClase
{
   public String unMetodo(
      int arreglo2[], int x, String salida )
   {
      if ( x < arreglo2.length )
         return String.format(
            "%s%d ", unMetodo( arreglo2, x + 1 ), arreglo2[ x ] );
      else
         return "";
   } // fin del método unMetodo
} // fin de la clase UnaClase
```

```
// Ejercicio 15.13 Solución: PruebaUnaClase.java

public class PruebaUnaClase
{
   public static void main( String args[] )
   {
      UnaClase objetoUnaClase = new UnaClase();

      int arreglo[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      String resultados =
         objetoUnaClase.unMetodo( arreglo, 0 );

      System.out.println( resultados );
   } // fin de main
} // fin de la clase PruebaUnaClase
```

15.14 *(Palíndromos)* Un palíndromo es una cadena que se escribe de la misma forma tanto al derecho como al revés. Algunos ejemplos de palíndromos son "radar", "reconocer" y (si se ignoran los espacios) "anita lava la tina". Escriba un método recursivo llamado `probarPalindromo`, que devuelva el valor `boolean` `true` si la cadena almacenada en el arreglo es un palíndromo, y `false` en caso contrario. El método debe ignorar espacios y puntuación en la cadena.

15.15 *(Ocho reinas)* Un buen acertijo para los fanáticos del ajedrez es el problema de las Ocho reinas, que se describe a continuación: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de forma que ninguna reina "ataque" a otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Por ejemplo, si se coloca una reina en la esquina superior izquierda del tablero, no pueden colocarse otras reinas en ninguna de las posiciones marcadas que se muestran en la figura 15.23. Resuelva el problema mediante el uso de recursividad. [*Sugerencia:* su solución debe empezar con la primera columna y buscar una ubicación en esa columna, en donde pueda colocarse una reina; al principio, coloque la reina en la primera fila. Después, la solución debe buscar en forma recursiva el resto de las columnas. En las primeras columnas, habrá varias ubicaciones en donde pueda colocarse una reina. Tome la primera posición disponible. Si se llega a una columna sin que haya una posible ubicación para una reina, el programa deberá regresar a la columna anterior y desplazar la reina que está en esa columna hacia una nueva fila. Este proceso continuo de retroceder y probar nuevas alternativas es un ejemplo de la "vuelta atrás" recursiva].

15.16 *(Imprimir un arreglo)* Escriba un método recursivo llamado `imprimirArreglo`, que muestre todos los elementos en un arreglo de enteros, separados por espacios.

15.17 *(Imprimir un arreglo al revés)* Escriba un método recursivo llamado `cadenaInversa`, que reciba un arreglo de caracteres que contenga una cadena como argumento, y que la imprima al revés. [*Sugerencia:* use el método `String` llamado `toCharArray`, el cual no recibe argumentos, para obtener un arreglo `char` que contenga los caracteres en el objeto `String`.]

15.18 *(Buscar el valor mínimo en un arreglo)* Escriba un método recursivo llamado `minimoRecursivo`, que determine el elemento más pequeño en un arreglo de enteros. Este método deberá regresar cuando reciba un arreglo de un elemento.

15.19 *(Fractales)* Repita el patrón del fractal de la sección 15.8 para formar una estrella. Empiece con cinco líneas en vez de una, en donde cada línea es un pico distinto de la estrella. Aplique el patrón del "fractal Lo" a cada pico de la estrella.

15.20 *(Recorrido de un laberinto mediante el uso de la "vuelta atrás" recursiva)* La cuadrícula que contiene caracteres `#` y puntos (`.`) en la figura 15.24 es una representación de un laberinto mediante un arreglo bidimensional. Los caracteres `#` representan las paredes del laberinto, y los puntos representan las ubicaciones en las posibles rutas a través del laberinto. Sólo pueden realizarse movimientos hacia una ubicación en el arreglo que contenga un punto.

Escriba un método recursivo (`recorridoLaberinto`) para avanzar a través de laberintos como el de la figura 15.24. El método debe recibir como argumentos un arreglo de caracteres de 12 por 12 que representa el laberinto, y la posición actual en el laberinto (la primera vez que se llama a este método, la posición actual debe ser el punto de entrada del laberinto). A medida que `recorridoLaberinto` trate de localizar la salida, debe colocar el carácter `x` en cada posición en la ruta. Hay un algoritmo simple para avanzar a través de un laberinto, que garantiza encontrar la salida (suponiendo que haya una). Si no hay salida, llegaremos a la posición inicial de nuevo. El algoritmo es el siguiente: partiendo de la posición actual en el laberinto, trate de avanzar un espacio en cualquiera de las posibles direcciones (abajo, derecha, arriba o izquierda). Si es posible avanzar por lo menos en una dirección, llame a `recorridoLaberinto` en forma recursiva, pasándole la nueva posición en el laberinto como la posición actual. Si no es posible avanzar en ninguna dirección, "retroceda" a una posición anterior en el laberinto y pruebe una nueva dirección para esa posición. Programe

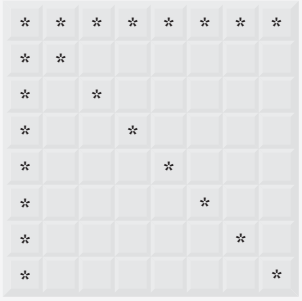


Figura 15.23 | Eliminación de posiciones al colocar una reina en la esquina superior izquierda de un tablero de ajedrez.

el método para que muestre el laberinto después de cada movimiento, de manera que el usuario pueda observar a la hora de que se resuelva el laberinto. La salida final del laberinto deberá mostrar sólo la ruta necesaria para resolverlo; si al ir en una dirección específica se llega a un punto sin salida, no se deben mostrar las `x` que avancen en esa dirección. [*Sugerencia:* para mostrar sólo la ruta final, tal vez sea útil marcar las posiciones que resulten en un punto sin salida con otro carácter (como `'O'`)].

15.21 *(Generación de laberintos al azar)* Escriba un método llamado `generadorLaberintos`, que reciba como argumento un arreglo bidimensional de 12 por 12 caracteres, y que produzca un laberinto al azar. Este método también deberá proporcionar las posiciones inicial y final del laberinto. Pruebe su método `recorridoLaberinto` del ejercicio 15.20, usando varios laberintos generados al azar.

15.22 *(Laberintos de cualquier tamaño)* Generalice los métodos `recorridoLaberinto` y `generadorLaberintos` de los ejercicios 15.20 y 15.21 para procesar laberintos de cualquier anchura y altura.

15.23 *(Tiempo para calcular números de Fibonacci)* Mejore el programa de Fibonacci de la figura 15.5, de manera que calcule el monto de tiempo aproximado requerido para realizar el cálculo, y el número de llamadas realizadas al método recursivo. Para este fin, llame al método `static` de `System` llamado `currentTimeMillis`, el cual no recibe argumento y devuelve el tiempo actual de la computadora en milisegundos. Llame a este método dos veces; una antes y la otra después de la llamada a `fibonacci`. Guarde cada valor y calcule la diferencia en los tiempos, para determinar cuántos milisegundos se requirieron para realizar el cálculo. Después, agregue una variable a la clase `CalculoFibonacci`, y utilice esta variable para determinar el número de llamadas realizadas al método `fibonacci`. Muestre sus resultados.

```
# # # # # # # # # # # # #
# . . . # . . . . . . . #
. . # . # . # # # # . # #
# # # . # . . . . # . # #
# . . . . # # # . # . . .
# # # # . # . # . # . # #
# . . # . # . # . # . # #
# # . # . # . # . # . # #
# . . . . . . . . # . # #
# # # # # # . # # # . # #
# . . . . . . # . . . # #
# # # # # # # # # # # # #
```

Figura 15.24 | Representación de un laberinto mediante un arreglo bidimensional.