23

Subprocesamiento múltiple

La definición más general de la belleza… múltiple deidad en la unidad.

—Samuel Taylor Coleridge

No bloquees el camino de la investigación.

—Charles Sanders Peirce

Una persona con un reloj sabe qué hora es; una persona con dos relojes nunca estará segura.

—Proverbio

Aprenda a laborar y esperar.

—Henry Wadsworth Longfellow

El mundo avanza tan rápido estos días que el hombre que dice que no puede hacer algo, por lo general, se ve interrumpido por alguien que lo está haciendo.

—Elbert Hubbard

OBJETIVOS

En este capítulo aprenderá a:

- Conocer qué son los subprocesos y por qué son útiles.
- Conocer cómo nos permiten los subprocesos administrar actividades concurrentes.
- Conocer el ciclo de vida de un subproceso.
- Conocer acerca de las prioridades y la programación de subprocesos.
- Crear y ejecutar objetos `Runnable`.
- Conocer acerca de la sincronización de subprocesos.
- Saber qué son las relaciones productor/consumidor y cómo se implementan con el subprocesamiento múltiple.
- Habilitar varios subprocesos para actualizar los componentes de GUI de Swing en forma segura para los subprocesos.
- Conocer acerca de las interfaces `Callable` y `Future`, que podemos usar para ejecutar tareas que devuelven resultados.

Plan general

23.1 Introducción

Sería bueno si pudiéramos concentrar nuestra atención en realizar una acción a la vez, y realizarla bien, pero por lo general, eso es difícil. El cuerpo humano realiza una gran variedad de operaciones en **paralelo**; o, como diremos en este capítulo, **operaciones concurrentes**. Por ejemplo, la respiración, la circulación de la sangre, la digestión, la acción de pensar y caminar pueden ocurrir de manera concurrente. Todos los sentidos (vista, tacto, olfato, gusto y oído) pueden emplearse al mismo tiempo. Las computadoras también pueden realizar operaciones concurrentes. Para las computadoras personales es tarea común compilar un programa, enviar un archivo a una impresora y recibir mensajes de correo electrónico a través de una red, de manera concurrente. Sólo las computadoras que tienen múltiples procesadores pueden realmente ejecutar varias instrucciones en forma concurrente. Los sistemas operativos en las computadoras con un solo procesador crean la ilusión de la ejecución concurrente, al alternar rápidamente entre una tarea y otra, pero en dichas computadoras sólo se puede ejecutar una instrucción a la vez.

La mayoría de los lenguajes de programación no permiten a los programadores especificar actividades concurrentes. Por lo general, los lenguajes proporcionan instrucciones de control secuenciales, las cuales permiten a los programadores especificar que sólo debe realizarse una acción a la vez, en donde la ejecución procede a la siguiente acción una vez que la anterior haya terminado. Históricamente, la concurrencia se ha implementado en forma de funciones primitivas del sistema operativo, disponibles sólo para los programadores de sistemas altamente experimentados.

El lenguaje de programación Ada, desarrollado por el Departamento de defensa de los Estados Unidos, hizo que las primitivas de concurrencia estuvieran disponibles ampliamente para los contratistas de defensa dedicados a la construcción de sistemas militares de comando y control. Sin embargo, Ada no se ha utilizado de manera general en las universidades o en la industria.

Java pone las primitivas de concurrencia a disposición del programador de aplicaciones, a través del lenguaje y de las APIs. El programador especifica que una aplicación contiene **subprocesos de ejecución** separados, en donde cada subproceso tiene su propia pila de llamadas a métodos y su propio contador, lo cual le permite ejecutarse concurrentemente con otros subprocesos, al mismo tiempo que comparte los recursos a nivel de aplicación (como la memoria) con estos otros subprocesos. Esta capacidad, llamada **subprocesamiento múltiple**, no está disponible en los lenguajes C y C++ básicos, los cuales influenciaron el diseño de Java.

Tip de rendimiento 23.1

Un problema con las aplicaciones de un solo subproceso es que las actividades que llevan mucho tiempo deben completarse antes de que puedan comenzar otras. En una aplicación con subprocesamiento múltiple, los subprocesos pueden distribuirse entre varios procesadores (si hay disponibles), de manera que se realicen varias tareas en forma concurrente, lo cual permite a la aplicación operar con más eficiencia. El subprocesamiento múltiple también puede incrementar el rendimiento en sistemas con un solo procesador que simulan la concurrencia; cuando un subproceso no puede proceder (debido a que, por ejemplo, está esperando el resultado de una operación de E/S), otro puede usar el procesador.

A diferencia de los lenguajes que no tienen capacidades integradas de subprocesamiento múltiple (como C y C++) y que, por lo tanto, deben realizar llamadas no portables a las primitivas de subprocesamiento múltiple del sistema operativo, Java incluye las primitivas de subprocesamiento múltiple como parte del mismo lenguaje y como parte de sus bibliotecas. Esto facilita la manipulación de los subprocesos de una manera portable entre plataformas.

Hablaremos sobre muchas aplicaciones de la **programación concurrente**. Por ejemplo, cuando se descarga un archivo extenso (como una imagen, un clip de audio o video) a través de Internet, tal vez el usuario no quiera esperar hasta que se descargue todo un clip completo para empezar a reproducirlo. Para resolver este problema podemos poner varios subprocesos a trabajar; uno para descargar el clip y otro para reproducirlo. Estas actividades se llevan a cabo concurrentemente. Para evitar que la reproducción del clip tenga interrupciones, **sincronizamos** (coordinamos las acciones de) los subprocesos de manera que el subproceso de reproducción no empiece sino hasta que haya una cantidad suficiente del clip en memoria, como para mantener ocupado al subproceso de reproducción.

La Máquina Virtual de Java (JVM) utiliza subprocesos también. Además de crear subprocesos para ejecutar un programa, la JVM también puede crear subprocesos para llevar a cabo tareas de mantenimiento, como la recolección de basura.

Escribir programas con subprocesamiento múltiple puede ser un proceso complicado. Aunque la mente humana puede realizar funciones de manera concurrente, a las personas se les dificulta saltar de un tren paralelo de pensamiento al otro. Para ver por qué los programas con subprocesamiento múltiple pueden ser difíciles de escribir y comprender, intente realizar el siguiente experimento: abra tres libros en la página 1 y trate de leerlos concurrentemente. Lea unas cuantas palabras del primer libro; después unas cuantas del segundo y por último otras cuantas del tercero. Luego, regrese a leer las siguientes palabras del primer libro, etcétera. Después de este experimento podrá apreciar muchos de los retos que implica el subprocesamiento múltiple: alternar entre los libros, leer brevemente, recordar en dónde se quedó en cada libro, acercar el libro que está leyendo para poder verlo, hacer a un lado los libros que no está leyendo y, entre todo este caos, ¡tratar de comprender el contenido de cada uno!

La programación de aplicaciones concurrentes es una tarea difícil y propensa a errores. Si descubre que debe usar la sincronización en un programa, debe seguir ciertos lineamientos simples. Primero, utilice las clases existentes de la API de Java (como la clase `ArrayBlockingQueue` que veremos en la sección 23.7, Relación productor/consumidor: `ArrayBlockingQueue`) que administren la sincronización por usted. Las clases en la API de Java están escritas por expertos, han sido probadas y depuradas por completo, operan con eficiencia y le ayudan a evitar trampas y obstáculos comunes. En segundo lugar, si descubre que necesita una funcionalidad más personalizada de la que se proporciona en las APIs de Java, debe usar la palabra clave `synchronized` y los métodos `wait`, `notify` y `notifyAll` de `Object` (que veremos en las secciones 23.5 y 23.8). Por último, si requiere herramientas aún más complejas, entonces debe usar las `interfaces Lock` y `Condition` que se presentan en la sección 23.10.

Sólo los programadores avanzados que estén familiarizados con las trampas y obstáculos comunes de la programación concurrente deben utilizar las `interfaces Lock` y `Condition`. En este capítulo explicaremos estos temas por varias razones: proporcionan una base sólida para comprender cómo las aplicaciones concurrentes sincronizan el acceso a la memoria compartida; los conceptos son importantes de comprender, aun si una aplicación no utiliza estas herramientas de manera explícita; y al demostrarle la complejidad involucrada en el uso de estas características de bajo nivel, esperamos dejar en usted la impresión acerca de la importancia del uso de las herramientas de concurrencia preempaquetadas, siempre que sea posible.

23.2 Estados de los subprocesos: ciclo de vida de un subproceso

En cualquier momento dado, se dice que un subproceso se encuentra en uno de varios **estados de subproceso** (los cuales se muestran en el diagrama de estados de UML de la figura 23.1). Varios de los términos en el diagrama se definen en secciones posteriores.

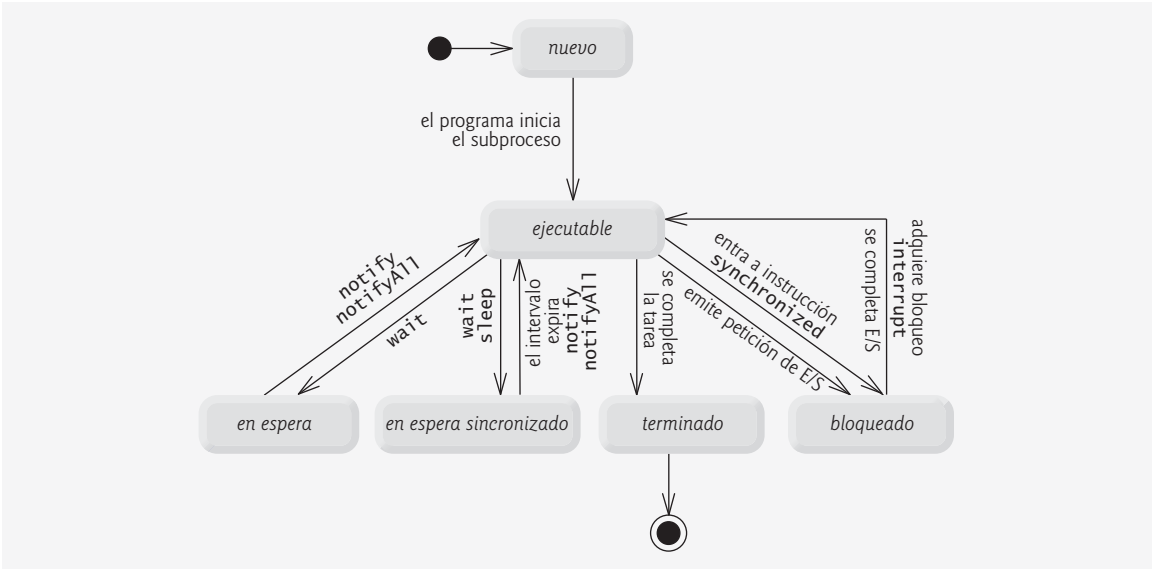


Figura 23.1 | Diagrama de estado de UML del ciclo de vida de un subproceso.

Un nuevo subproceso empieza su ciclo cuando hace la transición al estado ***nuevo***; permanece en este estado hasta que el programa inicia el subproceso, con lo cual se coloca en el estado ***ejecutable***. Se considera que un *subproceso* en el estado ejecutable está ejecutando su tarea.

Algunas veces, un subproceso *ejecutable* cambia al estado ***en espera*** mientras espera a que otro subproceso realice una tarea. Un subproceso *en espera* regresa al estado *ejecutable* sólo cuando otro subproceso notifica al subproceso esperando que puede continuar ejecutándose.

Un subproceso *ejecutable* puede entrar al estado ***en espera sincronizado*** durante un intervalo específico de tiempo. Regresa al estado *ejecutable* cuando ese intervalo de tiempo expira, o cuando ocurre el evento que está esperando. Los subprocesos en *espera sincronizado* y *en espera* no pueden usar un procesador, aun cuando haya uno disponible. Un subproceso *ejecutable* puede cambiar al estado *en espera sincronizado* si proporciona un intervalo de espera opcional cuando está esperando a que otro subproceso realice una tarea. Dicho subproceso regresa al estado *ejecutable* cuando se lo notifica otro subproceso, o cuando expira el intervalo de tiempo; lo que ocurra primero. Otra manera de colocar a un subproceso en el estado *en espera sincronizado* es dejar inactivo un subproceso *ejecutable*. Un **subproceso inactivo** permanece en el estado *en espera sincronizado* durante un periodo designado de tiempo (conocido como **intervalo de inactividad**), después del cual regresa al estado *ejecutable*. Los subprocesos están inactivos cuando, en cierto momento, no tienen una tarea que realizar. Por ejemplo, un procesador de palabras puede contener un subproceso que periódicamente respalde (es decir, escriba una copia de) el documento actual en el disco, para fines de recuperarlo. Si el subproceso no quedara inactivo entre un respaldo y otro, requeriría un ciclo en el que se evaluara continuamente si debe escribir una copia del documento en el disco. Este ciclo consumiría tiempo del procesador sin realizar un trabajo productivo, con lo cual se reduciría el rendimiento del sistema. En este caso, es más eficiente para el subproceso especificar un intervalo de inactividad (equivalente al periodo entre respaldos sucesivos) y entrar al estado *en espera sincronizado*. Este subproceso se regresa al estado *ejecutable* cuando expire su intervalo de inactividad, punto en el cual escribe una copia del documento en el disco y vuelve a entrar al estado *en espera sincronizado*.

Un subproceso *ejecutable* cambia al estado ***bloqueado*** cuando trata de realizar una tarea que no puede completarse inmediatamente, y debe esperar temporalmente hasta que se complete esa tarea. Por ejemplo, cuando un subproceso emite una petición de entrada/salida, el sistema operativo bloquea el subproceso para que no se ejecute sino hasta que se complete la petición de E/S; en ese punto, el subproceso *bloqueado* cambia al estado *ejecutable*, para poder continuar su ejecución. Un subproceso *bloqueado* no puede usar un procesador, aun si hay uno disponible.

Un subproceso ejecutable entra al estado ***terminado*** (algunas veces conocido como el estado ***muerto***) cuando completa exitosamente su tarea, o termina de alguna otra forma (tal vez debido a un error). En el diagrama

de estados de UML de la figura 23.1, el estado *terminado* va seguido por el estado final de UML (el símbolo de viñeta) para indicar el final de las transiciones de estado.

A nivel del sistema operativo, el estado *ejecutable* de Java generalmente abarca dos estados separados (figura 23.2). El sistema operativo oculta estos estados de la Máquina Virtual de Java (JVM), la cual sólo ve el estado *ejecutable*. Cuando un subproceso cambia por primera vez al estado *ejecutable* desde el estado *nuevo*, el subproceso se encuentra en el estado ***listo***. Un subproceso *listo* entra al estado ***en ejecución*** (es decir, empieza su ejecución) cuando el sistema operativo lo asigna a un procesador; a esto también se le conoce como **despachar el subproceso**. En la mayoría de los sistemas operativos, a cada subproceso se le otorga una pequeña cantidad de tiempo del procesador (lo cual se conoce como ***quantum*** o **intervalo de tiempo**) en la que debe realizar su tarea. (El proceso de decidir qué tan grande debe ser el quantum de tiempo es un tema clave en los cursos de sistemas operativos). Cuando expira su quantum, el subproceso regresa al estado *listo* y el sistema operativo asigna otro subproceso al procesador (vea la sección 23.3). Las transiciones entre los estados *listo* y en *ejecución* las maneja únicamente el sistema operativo. La JVM no "ve" las transiciones; simplemente ve el subproceso como *ejecutable* y deja al sistema operativo la decisión de cambiar el subproceso entre *listo* y *ejecutable*. El proceso que utiliza un sistema operativo para determinar qué subproceso debe despachar se conoce como **programación de subprocesos**, y depende de las prioridades de los subprocesos (que veremos en la siguiente sección).

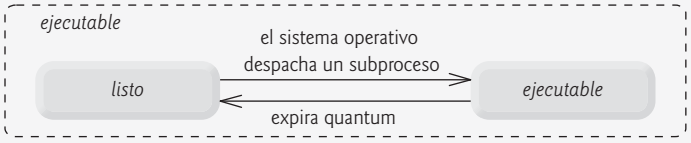


Figura 23.2 | Vista interna del sistema operativo del estado *ejecutable* de Java.

23.3 Prioridades y programación de subprocesos

Todo subproceso en Java tiene una **prioridad de subproceso**, la cual ayuda al sistema operativo a determinar el orden en el que se programan los subprocesos. Las prioridades de Java varían entre `MIN_PRIORITY` (una constante de 1) y `MAX_PRIORITY` (una constante de 10). De manera predeterminada, cada subproceso recibe la prioridad `NORM_PRIORITY` (una constante de 5). Cada nuevo subproceso hereda la prioridad del subproceso que lo creó. De manera informal, los subprocesos de mayor prioridad son más importantes para un programa, y se les debe asignar tiempo del procesador antes que a los subprocesos de menor prioridad. Sin embargo, las prioridades de los subprocesos no garantizan el orden en el que se ejecutan los subprocesos.

[*Nota:* las constantes (`MAX_PRIORITY`, `MIN_PRIORITY` y `NORM_PRIORITY`) se declaran en la clase `Thread`. Se recomienda no crear y usar explícitamente objetos `Thread` para implementar la concurrencia, sino utilizar mejor la interfaz `Executor` (que describiremos en la sección 23.4.2). La clase `Thread` contiene varios métodos `static` útiles, los cuales describiremos más adelante en este capítulo].

La mayoría de los sistemas operativos soportan los intervalos de tiempo (timeslicing), que permiten a los subprocesos de igual prioridad compartir un procesador. Sin el intervalo de tiempo, cada subproceso en un conjunto de subprocesos de igual prioridad se ejecuta hasta completarse (a menos que deje el estado *ejecutable* y entre al estado *en espera* o en *espera sincronizado*, o lo interrumpa un subproceso de mayor prioridad) antes que otros subprocesos de igual prioridad tengan oportunidad de ejecutarse. Con el intervalo de tiempo, aun si un subproceso no ha terminado de ejecutarse cuando expira su quantum, el procesador se quita del subproceso y pasa al siguiente subproceso de igual prioridad, si hay uno disponible.

El **programador de subprocesos** de un sistema operativo determina cuál subproceso se debe ejecutar a continuación. Una implementación simple del programador de subprocesos mantiene el subproceso de mayor prioridad en *ejecución* en todo momento y, si hay más de un subproceso de mayor prioridad, asegura que todos esos subprocesos se ejecuten durante un quantum cada uno, en forma **cíclica (round-robin)**. En la figura 23.3 se muestra una **cola de prioridades multinivel** para los subprocesos. En la figura, suponiendo que hay una computadora con un solo procesador, los subprocesos A y B se ejecutan cada uno durante un quantum, en forma cíclica hasta que ambos subprocesos terminan su ejecución. Esto significa que A obtiene un quantum de tiempo

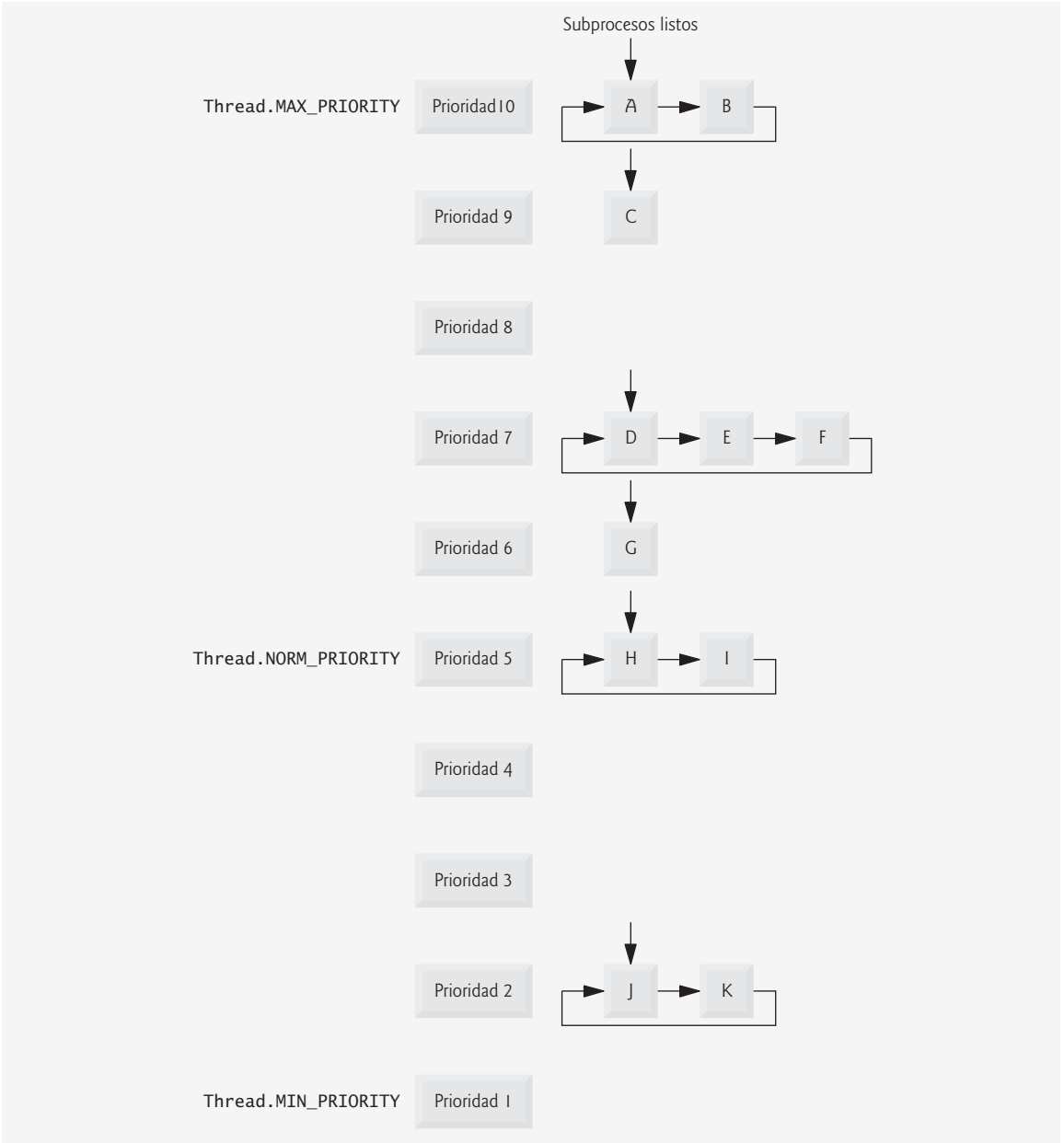


Figura 23.3 | Programación de prioridad de subprocesos.

para ejecutarse. Después B obtiene un quantum. Luego A obtiene otro quantum. Después B obtiene otro. Esto continúa hasta que un subproceso se completa. Entonces, el procesador dedica todo su poder al subproceso restante (a menos que esté listo otro subproceso de prioridad 10). A continuación, el subproceso C se ejecuta hasta completarse (suponiendo que no llegan subprocesos de mayor prioridad). Los subprocesos D, E y F se ejecutan cada uno durante un quantum, en forma cíclica hasta que todos terminan de ejecutarse (de nuevo, suponiendo que no llegan procesos de mayor prioridad). Este proceso continúa hasta que todos los subprocesos se ejecutan hasta completarse.

Cuando un subproceso de mayor prioridad entra al estado *listo*, el sistema operativo generalmente sustituye el subproceso actual en *ejecución* para dar preferencia al subproceso de mayor prioridad (una operación conocida como **programación preferente**). Dependiendo del sistema operativo, los subprocesos de mayor prioridad

podrían posponer (tal vez de manera indefinida) la ejecución de los subprocesos de menor prioridad. Comúnmente, a dicho **aplazamiento indefinido** se le conoce, en forma más colorida, como **inanición**.

Java cuenta con herramientas de concurrencia de alto nivel para ocultar parte de esta complejidad, y hacer que los programas sean menos propensos a errores. Las prioridades de los subprocesos se utilizan en segundo plano para interactuar con el sistema operativo, pero la mayoría de los programadores que utilizan subprocesamiento múltiple en Java no tienen que preocuparse por configurar y ajustar las prioridades de los subprocesos.

Tip de portabilidad 23.1

La programación de subprocesos es independiente de la plataforma; el comportamiento de un programa con subprocesamiento múltiple podría variar entre las distintas implementaciones de Java.

Tip de portabilidad 23.2

Al diseñar programas con subprocesamiento múltiple, considere las capacidades de subprocesamiento de las plataformas en las que se ejecutarán esos programas. Si utiliza prioridades distintas a las predeterminadas, el comportamiento de sus programas será específico para la plataforma en la que se ejecuten. Si su meta es la portabilidad, no ajuste las prioridades de los subprocesos.

23.4 Creación y ejecución de subprocesos

El medio preferido de crear aplicaciones en Java con subprocesamiento múltiple es mediante la implementación de la interfaz `Runnable` (del paquete `java.lang`). Un objeto `Runnable` representa una "tarea" que puede ejecutarse concurrentemente con otras tareas. La interfaz `Runnable` declara un solo método, `run`, el cual contiene el código que define la tarea que debe realizar un objeto `Runnable`. Cuando se crea e inicia un subproceso que ejecuta un objeto `Runnable`, el subproceso llama al método `run` del objeto `Runnable`, el cual se ejecuta en el nuevo subproceso.

23.4.1 Objetos `Runnable` y la clase `Thread`

La clase `TareaImprimir` (figura 23.4) implementa a `Runnable` (línea 5), de manera que puedan ejecutarse varios objetos `TareaImprimir` en forma concurrente. La variable `tiempoInactividad` (línea 7) almacena un valor entero aleatorio de 0 a 5 segundos, que se crea en el constructor de `TareaImprimir` (línea 16). Cada subproceso que ejecuta a un objeto `TareaImprimir` permanece inactivo durante la cantidad de tiempo especificado por `tiempoInactividad`, después imprime el nombre de la tarea y un mensaje indicando que terminó su inactividad.

```
1  // Fig. 23.4: TareaImprimir.java
2  // La clase TareaImprimir permanece inactiva durante un tiempo aleatorio entre 0 y 5
   segundos
3  import java.util.Random;
4
5  public class TareaImprimir implements Runnable
6  {
7     private final int tiempoInactividad; // tiempo de inactividad aleatorio para el
      subproceso
8     private final String nombreTarea; // nombre de la tarea
9     private final static Random generador = new Random();
10
11    public TareaImprimir( String nombre )
12    {
13       nombreTarea = nombre; // establece el nombre de la tarea
14
15       // elige un tiempo de inactividad aleatorio entre 0 y 5 segundos
16       tiempoInactividad = generador.nextInt( 5000 ); // milisegundos
```

Figura 23.4 | La clase `TareaImprimir` permanece inactiva durante un tiempo aleatorio de 0 a 5 segundos. (Parte 1 de 2).

```
17     } // fin del constructor de TareaImprimir
18
19     // el método run contiene el código que ejecutará un subproceso
20     public void run()
21     {
22        try // deja el subproceso inactivo durante tiempoInactividad segundos
23        {
24           System.out.printf( "%s va a estar inactivo durante %d milisegundos.\n",
25              nombreTarea, tiempoInactividad );
26           Thread.sleep( tiempoInactividad ); // deja el subproceso inactivo
27        } // fin de try
28        catch ( InterruptedException excepcion )
29        {
30           System.out.printf( "%s %s\n", nombreTarea,
31              "termino en forma prematura, debido a la interrupcion" );
32        } // fin de catch
33
34        // imprime el nombre de la tarea
35        System.out.printf( "%s termino su inactividad\n", nombreTarea );
36     } // fin del método run
37  } // fin de la clase TareaImprimir
```

Figura 23.4 | La clase `TareaImprimir` permanece inactiva durante un tiempo aleatorio de 0 a 5 segundos. (Parte 2 de 2).

Un objeto `TareaImprimir` se ejecuta cuando un subproceso llama al método `run` de `TareaImprimir`. En las líneas 24 y 25 se muestra un mensaje, indicando el nombre de la tarea actual en ejecución y que ésta va a quedar inactiva durante `tiempoInactividad` milisegundos. En la línea 26 se invoca al método `static` `sleep` de la clase `Thread`, para colocar el subproceso en el estado *en espera sincronizado* durante la cantidad especificada de tiempo. En este punto, el subproceso pierde al procesador y el sistema permite que otro subproceso se ejecute. Cuando el subproceso despierta, vuelve a entrar al estado *ejecutable*. Cuando el objeto `TareaImprimir` se asigna a otro procesador de nuevo, en la línea 35 se imprime un mensaje indicando que la tarea dejó de estar inactiva, y después el método `run` termina. Observe que la instrucción `catch` en las líneas 28 a 32 es obligatoria, ya que el método `sleep` podría lanzar una excepción **`InterruptedException`** (verificada) si se hace una llamada al método **`interrupt`** del subproceso inactivo.

En la figura 23.5 se crean objetos `Thread` para ejecutar objetos `TareaImprimir`. En las líneas 12 a 14 se crean tres subprocesos, cada uno de los cuales especifica un nuevo objeto `TareaImprimir` a ejecutar. En las líneas 19 a 21 se hace una llamada al método `start` de `Thread` en cada uno de los subprocesos; cada llamada invoca al método `run` del objeto `Runnable` correspondiente para ejecutar la tarea. En la línea 23 se imprime un mensaje indicando que se han iniciado todas las tareas de los subprocesos, y que el método `main` terminó su ejecución.

```
1  // Fig. 23.5: CreadorSubproceso.java
2  // Creación e inicio de tres subprocesos para ejecutar objetos Runnable.
3  import java.lang.Thread;
4
5  public class CreadorSubproceso
6  {
7     public static void main( String[] args )
8     {
9        System.out.println( "Creacion de subprocesos" );
10
11       // crea cada subproceso con un nuevo objeto Runnable
12       Thread subproceso1 = new Thread( new TareaImprimir( "tarea1" ) );
13       Thread subproceso2 = new Thread( new TareaImprimir( "tarea2" ) );
14       Thread subproceso3 = new Thread( new TareaImprimir( "tarea3" ) );
```

Figura 23.5 | Creación e inicio de tres subprocesos para ejecutar objetos `Runnable`. (Parte 1 de 2).

```
15
16         System.out.println( "Subprocesos creados, iniciando tareas." );
17
18         // inicia los subprocesos y los coloca en el estado "en ejecución"
19         subproceso1.start(); // invoca al método run de tarea1
20         subproceso2.start(); // invoca al método run de tarea2
21         subproceso3.start(); // invoca al método run de tarea3
22
23         System.out.println( "Tareas iniciadas, main termina.\n" );
24      } // fin de main
25   } // fin de la clase CreadorSubproceso
```

```
Creacion de subprocesos
Subprocesos creados, iniciando tareas
Tareas iniciadas, main termina

Tarea3 va a estar inactivo durante 491 milisegundos
tarea2 va a estar inactivo durante 71 milisegundos
Tarea1 va a estar inactivo durante 3549 milisegundos
tarea2 termino su inactividad
tarea3 termino su inactividad
tarea1 termino su inactividad
```

```
Creacion de subprocesos
Subprocesos creados, iniciando tareas
tarea1 va a estar inactivo durante 4666 milisegundos
tarea2 va a estar inactivo durante 48 milisegundos
tarea3 va a estar inactivo durante 3924 milisegundos
Tareas iniciadas, main termina

subproceso2 termino su inactividad
subproceso3 termino su inactividad
subproceso1 termino su inactividad
```

Figura 23.5 | Creación e inicio de tres subprocesos para ejecutar objetos `Runnable`. (Parte 2 de 2).

El código en el método `main` se ejecuta en el **subproceso principal**, un subproceso creado por la JVM. El código en el método `run` de `TareaImprimir` (líneas 20 a 36 de la figura 23.4) se ejecuta en los subprocesos creados en las líneas 12 a 14 de la figura 23.5. Cuando termina el método `main`, el programa en sí continúa ejecutándose, ya que aún hay subprocesos vivos (es decir, alguno de los tres subprocesos que creamos no ha llegado aún al estado *terminado*). El programa no terminará sino hasta que su último subproceso termine de ejecutarse, punto en el cual la JVM también terminará.

Los resultados de ejemplo para este programa muestran el nombre de cada tarea y el tiempo de inactividad, al momento en que el subproceso queda inactivo. El subproceso con el tiempo de inactividad más corto es el que normalmente se despierta primero, indicando que acaba de dejar de estar inactivo y termina. En la sección 23.9 hablaremos sobre las cuestiones acerca del subprocesamiento múltiple que podrían evitar que el subproceso con el tiempo de inactividad más corto se despertara primero. En el primer conjunto de resultados, el subproceso principal termina antes de que cualquiera de los otros subprocesos impriman sus nombres y tiempos de inactividad. Esto muestra que el subproceso principal se ejecuta hasta completarse antes que cualquiera de los otros subprocesos tengan la oportunidad de ejecutarse. En el segundo conjunto de resultados, todos los subprocesos imprimen sus nombres y tiempos de inactividad antes de que termine el subproceso principal. Esto muestra que el sistema operativo permitió a los otros subprocesos ejecutarse antes de que terminara el subproceso principal. Éste es un ejemplo de la programación cíclica (round-robin) que vimos en la sección 23.3. Además, observe que en el primer conjunto de resultados de ejemplo, `tarea3` queda inactivo primero y `tarea1` queda inactivo al último, aun cuando empezamos el subproceso de `tarea1` primero. Esto ilustra el hecho de que no podemos predecir el orden en el que se programarán los subprocesos, aun si conocemos el orden en el que se crearon e iniciaron. Por

último, en cada uno de los conjuntos de resultados, observe que el orden en el que los subprocesos indican que dejaron de estar inactivos coincide con los tiempos de inactividad de menor a mayor de los tres subprocesos. Aunque éste es el orden razonable e intuitivo para que estos subprocesos terminen sus tareas, no se garantiza que los subprocesos vayan a terminar en este orden.

23.4.2 Administración de subprocesos con el marco de trabajo `Executor`

Aunque es posible crear subprocesos en forma explícita, como en la figura 23.5, se recomienda utilizar la interfaz `Executor` para administrar la ejecución de objetos `Runnable` de manera automática. Por lo general, un objeto **`Executor`** crea y administra un grupo de subprocesos, al cual se le denomina **reserva de subprocesos**, para ejecutar objetos `Runnable`. Usted puede crear sus propios subprocesos, pero el uso de un objeto `Executor` tiene muchas ventajas. Los objetos `Executor` pueden reutilizar los subprocesos existentes para eliminar la sobrecarga de crear un nuevo subproceso para cada tarea, y pueden mejorar el rendimiento al optimizar el número de subprocesos, con lo cual se asegura que el procesador se mantenga ocupado, sin crear demasiados subprocesos como para que la aplicación se quede sin recursos.

La interfaz `Executor` declara un solo método llamado **`execute`**, el cual acepta un objeto `Runnable` como argumento. El objeto `Executor` asigna cada objeto `Runnable` que se pasa a su método `execute` a uno de los subprocesos disponibles en la reserva. Si no hay subprocesos disponibles, el objeto `Executor` crea un nuevo subproceso, o espera a que haya uno disponible y lo asigna al objeto `Runnable` que se pasó al método `execute`.

La interfaz **`ExecutorService`** (del paquete `java.util.concurrent`) es una interfaz que extiende a `Executor` y declara varios métodos más para administrar el ciclo de vida de un objeto `Executor`. Un objeto que implementa a la interfaz `ExecutorService` se puede crear mediante el uso de los métodos `static` declarados en la clase `Executors` (del paquete `java.util.concurrent`). Utilizaremos la interfaz `ExecutorService` y un método de la clase **`Executors`** en la siguiente aplicación, la cual ejecuta tres tareas.

En la figura 23.6 se utiliza un objeto `ExecutorService` para administrar subprocesos que ejecuten objetos `TareaImprimir`. El método `main` (líneas 8 a 29) crea y nombra tres objetos `TareaImprimir` (líneas 11 a 13). En la línea 18 se utiliza el método **`newCachedThreadPool`** de `Executors` para obtener un objeto `ExecutorService` que crea nuevos subprocesos, según los va necesitando la aplicación. Estos subprocesos los utiliza `ejecutorSubprocesos` para ejecutar los objetos `Runnable`.

```
1  // Fig. 23.6: EjecutorTareas.java
2  // Uso de un objeto ExecutorService para ejecutar objetos Runnable.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5
6  public class EjecutorTareas
7  {
8     public static void main( String[] args )
9     {
10       // crea y nombra cada objeto Runnable
11       TareaImprimir tarea1 = new TareaImprimir( "tarea1" );
12       TareaImprimir tarea2 = new TareaImprimir( "tarea2" );
13       TareaImprimir tarea3 = new TareaImprimir( "tarea3" );
14
15       System.out.println( "Iniciando Executor" );
16
17       // crea objeto ExecutorService para administrar los subprocesos
18       ExecutorService ejecutorSubprocesos = Executors.newCachedThreadPool();
19
20       // inicia los subprocesos y los coloca en el estado ejecutable
21       ejecutorSubprocesos.execute( tarea1 ); // inicia tarea1
22       ejecutorSubprocesos.execute( tarea2 ); // inicia tarea2
23       ejecutorSubprocesos.execute( tarea3 ); // inicia tarea3
24
```

Figura 23.6 | Uso de un objeto `ExecutorService` para ejecutar objetos `Runnable`. (Parte 1 de 2).

```
25         // cierra los subprocesos trabajadores cuando terminan sus tareas
26         ejecutorSubprocesos.shutdown();
27
28         System.out.println( "Tareas iniciadas, main termina.\n" );
29      } // fin de main
30   } // fin de la clase EjecutorTareas
```

```
Iniciando Executor
Tareas iniciadas, main termina

tarea1 va a estar inactivo durante 4806 milisegundos
tarea2 va a estar inactivo durante 2513 milisegundos
tarea3 va a estar inactivo durante 1132 milisegundos
tarea3 termino su inactividad
tarea2 termino su inactividad
tarea1 termino su inactividad
```

```
Iniciando Executor
tarea1 va a estar inactivo durante 1342 milisegundos
tarea2 va a estar inactivo durante 277 milisegundos
tarea3 va a estar inactivo durante 2737 milisegundos
Tareas iniciadas, main termina

subproceso2 termino su inactividad
subproceso1 termino su inactividad
subproceso3 termino su inactividad
```

Figura 23.6 | Uso de un objeto `ExecutorService` para ejecutar objetos `Runnable`. (Parte 2 de 2).

En cada una de las líneas 21 a 23 se invoca el método `execute` de `ExecutorService`. Este método ejecuta el objeto `Runnable` que recibe como argumento (en este caso, un objeto `TareaImprimir`) en algún momento en el futuro. La tarea especificada puede ejecutarse en uno de los subprocesos en la reserva de `ExecutorService`, en un nuevo subproceso creado para ejecutarla, o en el subproceso que llamó al método `execute`; el objeto `ExecutorService` administra estos detalles. El método `execute` regresa inmediatamente después de cada invocación; el programa no espera a que termine cada objeto `TareaImprimir`. En la línea 26 se hace una llamada al método **`shutdown`** de `ExecutorService`, el cual notifica al objeto `ExecutorService` para que deje de aceptar nuevas tareas, pero continúa ejecutando las tareas que ya se hayan enviado. Una vez que se han completado todos los objetos `Runnable` enviados anteriormente, el objeto `ejecutorSubprocesos` termina. En la línea 28 se imprime un mensaje indicando que se iniciaron las tareas y que el subproceso principal está terminando su ejecución. Los conjuntos de resultados de ejemplo son similares a los del programa anterior y, de nuevo, demuestran el no determinismo de la programación de subprocesos.

23.5 Sincronización de subprocesos

Cuando varios subprocesos comparten un objeto, y éste puede ser modificado por uno o más de los subprocesos, pueden ocurrir resultados indeterminados (como veremos en los ejemplos), a menos que el acceso al objeto compartido se administre de manera apropiada. Si un subproceso se encuentra en el proceso de actualizar a un objeto compartido, y otro subproceso trata de actualizarlo también, no queda claro cuál actualización del subproceso se lleva a cabo. Cuando esto ocurre, el comportamiento del programa no puede determinarse; algunas veces el programa producirá los resultados correctos; sin embargo, en otras ocasiones producirá los resultados incorrectos. En cualquier caso, no habrá indicación de que el objeto compartido se manipuló en forma incorrecta.

El problema puede resolverse si se da a un subproceso a la vez el *acceso exclusivo* al código que manipula al objeto compartido. Durante ese tiempo, otros subprocesos que deseen manipular el objeto deben mantenerse en espera. Cuando el subproceso con acceso exclusivo al objeto termina de manipularlo, a uno de los subprocesos que estaba en espera se le debe permitir que continúe ejecutándose. Este proceso, conocido como **sincronización de subprocesos**, coordina el acceso a los datos compartidos por varios subprocesos concurrentes. Al sincronizar

los subprocesos de esta forma, podemos asegurar que cada subproceso que accede a un objeto compartido excluye a los demás subprocesos de hacerlo en forma simultánea; a esto se le conoce como **exclusión mutua**.

Una manera de realizar la sincronización es mediante los **monitores** integrados en Java. Cada objeto tiene un monitor y un **bloqueo de monitor** (o **bloqueo intrínseco**). El monitor asegura que el bloqueo de monitor de su objeto se mantenga por un máximo de sólo un subproceso a la vez. Así, los monitores y los bloqueos de monitor se pueden utilizar para imponer la exclusión mutua. Si una operación requiere que el subproceso en ejecución mantenga un bloqueo mientras se realiza la operación, un subproceso debe adquirir el bloqueo para poder continuar con la operación. Otros subprocesos que traten de realizar una operación que requiera el mismo bloqueo permanecerán bloqueados hasta que el primer subproceso libere el *bloqueo*, punto en el cual los subprocesos bloqueados pueden tratar de adquirir el *bloqueo* y continuar con la operación.

Para especificar que un subproceso debe mantener un bloqueo de monitor para ejecutar un bloque de código, el código debe colocarse en una **instrucción `synchronized`**. Se dice que dicho código está **protegido** por el bloqueo de monitor; un subproceso debe **adquirir el bloqueo** para ejecutar las instrucciones `synchronized`. El monitor sólo permite que un subproceso a la vez ejecute instrucciones dentro de bloques `synchronized` que se bloqueen en el mismo objeto, ya que sólo un subproceso a la vez puede mantener el bloqueo de monitor. Las instrucciones `synchronized` se declaran mediante la **palabra clave `synchronized`**:

```
synchronized ( objeto )
{
    Instrucciones
} // fin de la instrucción synchronized
```

en donde objeto es el *objeto* cuyo bloqueo de monitor se va a adquirir; generalmente, *objeto* es `this` si es el objeto en el que aparece la instrucción `synchronized`. Si varias instrucciones `synchronized` están tratando de ejecutarse en un objeto al mismo tiempo, sólo una de ellas puede estar activa en el objeto; todos los demás subprocesos que traten de entrar a una instrucción `synchronized` en el mismo objeto se colocan en el estado *bloqueado*.

Cuando una instrucción `synchronized` termina de ejecutarse, el bloqueo de monitor del objeto se libera y el sistema operativo puede permitir que uno de los subprocesos *bloqueados*, que intentan entrar a una instrucción `synchronized`, adquieran el bloqueo para continuar. Java también permite los **métodos `synchronized`**. Dicho método es equivalente a una instrucción `synchronized` que encierra el cuerpo completo de un método, y que utiliza a `this` como el objeto cuyo bloqueo de monitor se va a adquirir. Puede especificar un método como `synchronized`; para ello, coloque la palabra clave `synchronized` antes del tipo de valor de retorno del método en su declaración.

23.5.1 Cómo compartir datos sin sincronización

Ahora presentaremos un ejemplo para ilustrar los peligros de compartir un objeto entre varios subprocesos sin una sincronización apropiada. En este ejemplo, dos objetos `Runnable` mantienen referencias a un solo arreglo entero. Cada objeto `Runnable` escribe cinco valores en el arreglo, y después termina. Tal vez esto parezca inofensivo, pero puede provocar errores si el arreglo se manipula sin sincronización.

La clase `ArregloSimple`

Un objeto de la clase `ArregloSimple` (figura 23.7) se compartirá entre varios subprocesos. `ArregloSimple` permitirá que esos subprocesos coloquen valores `int` en `arreglo` (declarado en la línea 7). En la línea 8 se inicializa la variable `indiceEscritura`, la cual se utilizará para determinar el elemento del arreglo en el que se debe escribir a continuación. El constructor (líneas 12 a 15) crea un arreglo entero del tamaño deseado.

```
1  // Fig. 23.7: ArregloSimple.java
2  // Clase que administra un arreglo simple para compartirlo entre varios subprocesos.
3  import java.util.Random;
4
5  public class ArregloSimple // PRECAUCIÓN: ¡NO ES SEGURO PARA LOS SUBPROCESOS!
6  {
```

Figura 23.7 | Clase que administra un arreglo entero para compartirlo entre varios subprocesos. (Parte 1 de 2).

```
 7     private final int arreglo[]; // el arreglo entero compartido
 8     private int indiceEscritura = 0; // índice del siguiente elemento a escribir
 9     private final static Random generador = new Random();
10
11     // construye un objeto ArregloSimple de un tamaño dado
12     public ArregloSimple( int tamanio )
13     {
14        arreglo = new int[ tamanio ];
15     } // fin del constructor
16
17     // agrega un valor al arreglo compartido
18     public void agregar( int valor )
19     {
20        int posicion = indiceEscritura; // almacena el índice de escritura
21
22        try
23        {
24           // pone el subproceso en inactividad de 0 a 499 milisegundos
25           Thread.sleep( generador.nextInt( 500 ) );
26        } // fin de try
27        catch ( InterruptedException ex )
28        {
29           ex.printStackTrace();
30        } // fin de catch
31
32        // coloca el valor en el elemento apropiado
33        arreglo[ posicion ] = valor;
34        System.out.printf( "%s escribio %2d en el elemento %d.\n",
35           Thread.currentThread().getName(), valor, posicion );
36
37        ++indiceEscritura; // incrementa el índice del siguiente elemento a escribir
38        System.out.printf( "Siguiente indice de escritura: %d\n", indiceEscritura );
39     } // fin del método agregar
40
41     // se utiliza para imprimir el contenido del arreglo entero compartido
42     public String toString()
43     {
44        String cadenaArreglo = "\nContenido de ArregloSimple:\n";
45
46        for ( int i = 0; i < arreglo.length; i++ )
47           cadenaArreglo += arreglo[ i ] + " ";
48
49        return cadenaArreglo;
50     } // fin del método toString
51  } // fin de la clase ArregloSimple
```

Figura 23.7 | Clase que administra un arreglo entero para compartirlo entre varios subprocesos. (Parte 2 de 2).

El método `agregar` (líneas 18 a 39) permite insertar nuevos valores al final del arreglo. En la línea 20 se almacena el valor de `indiceEscritura` actual. En la línea 25, el subproceso que invoca a `agregar` queda inactivo durante un intervalo aleatorio de 0 a 499 milisegundos. Esto se hace para que los problemas asociados con el acceso desincronizado a los datos compartidos sean más obvios. Una vez que el subproceso deja de estar inactivo, en la línea 33 se inserta el valor que se pasa a agregar en el arreglo, en el elemento especificado por `posicion`. En las líneas 34 y 35 se imprime un mensaje, indicando el nombre del subproceso en ejecución, el valor que se insertó en el arreglo y en dónde se insertó. En la línea 37 se incrementa `indiceEscritura`, de manera que la siguiente llamada a `agregar` insertará un valor en el siguiente elemento del arreglo. En las líneas 42 a 50 se sobrescribe el método `toString` para crear una representación `String` del contenido del arreglo.

La clase `EscritorArreglo`

La clase `EscritorArreglo` (figura 23.8) implementa a la interfaz `Runnable` para definir una tarea para insertar valores en un objeto `ArregloSimple`. El constructor (líneas 10 a 14) recibe dos argumentos: un valor entero, que viene siendo el primer `valor` que insertará esta tarea en el objeto `ArregloSimple`, y una referencia al objeto `ArregloSimple`. En la línea 20 se invoca el método `agregar` en el objeto `ArregloSimple`. La tarea se completa una vez que se agregan tres enteros consecutivos, empezando con `valorInicial`, al objeto `ArregloSimple`.

```
1  // Fig. 23.8: EscritorArreglo.java
2  // Agrega enteros a un arreglo compartido con otros objetos Runnable
3  import java.lang.Runnable;
4
5  public class EscritorArreglo implements Runnable
6  {
7     private final ArregloSimple arregloSimpleCompartido;
8     private final int valorInicial;
9
10    public EscritorArreglo( int valor, ArregloSimple arreglo )
11    {
12       valorInicial = valor;
13       arregloSimpleCompartido= arreglo;
14    } // fin del constructor
15
16    public void run()
17    {
18       for ( int i = valorInicial; i < valorInicial + 3; i++ )
19       {
20          arregloSimpleCompartido.agregar( i ); // agrega un elemento al arreglo compartido
21       } // fin de for
22    } // fin del método run
23 } // fin de la clase EscritorArreglo
```

Figura 23.8 | Agrega enteros a un arreglo compartido con otros objetos `Runnable`.

La clase `PruebaArregloCompartido`

La clase `PruebaArregloCompartido` ejecuta dos tareas `EscritorArreglo` que agregan valores a un solo objeto `ArregloSimple`. En la línea 12 se construye un objeto `ArregloSimple` con seis elementos. En las líneas 15 y 16 se crean dos nuevas tareas `EscritorArreglo`, una que coloca los valores 1 a 3 en el objeto `ArregloSimple`, y una que coloca los valores 11 a 13. En las líneas 19 a 21 se crea un objeto `ExecutorService` y se ejecutan los dos objetos `EscritorArreglo`. En la línea 23 se invoca el método `shutDown` de `ExecutorService` para evitar que se inicien tareas adicionales, y para permitir que la aplicación termine cuando las tareas actuales en ejecución se completen.

```
1  // Fig 23.9: PruebaArregloCompartido.java
2  // Ejecuta dos objetos Runnable para agregar elementos a un objeto ArregloSimple
   compartido.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.TimeUnit;
6
7  public class PruebaArregloCompartido
8  {
9     public static void main( String[] arg )
10    {
```

Figura 23.9 | Ejecuta dos objetos `Runnable` para insertar valores en un arreglo compartido. (Parte 1 de 2).

```
11         // construye el objeto compartido
12         ArregloSimple arregloSimpleCompartido = new ArregloSimple( 6 );
13
14         // crea dos tareas para escribir en el objeto ArregloSimple compartido
15         EscritorArreglo escritor1 = new EscritorArreglo( 1, arregloSimpleCompartido );
16         EscritorArreglo escritor2 = new EscritorArreglo( 11, arregloSimpleCompartido );
17
18         // ejecuta las tareas con un objeto ExecutorService
19         ExecutorService ejecutor = Executors.newCachedThreadPool();
20         ejecutor.execute( escritor1 );
21         ejecutor.execute( escritor2 );
22
23         ejecutor.shutdown();
24
25         try
26         {
27            // espera 1 minuto para que ambos escritores terminen de ejecutarse
28            boolean tareasTerminaron = ejecutor.awaitTermination(
29               1, TimeUnit.MINUTES );
30
31            if ( tareasTerminaron )
32               System.out.println( arregloSimpleCompartido ); // imprime el contendio
33            else
34               System.out.println(
35                   "Se agoto el tiempo esperando a que las tareas terminaran." );
36         } // fin de try
37         catch ( InterruptedException ex )
38         {
39            System.out.println(
40                "Hubo una interrupcion mientras esperaba a que terminaran las tareas." );
41         } // fin de catch
42      } // fin de main
43   } // fin de la clase PruebaArregloCompartido
```

```
pool-1-thread-1 escribio  1 en el elemento 0.
Siguiente indice de escritura: 1
pool-1-thread-1 escribio  2 en el elemento 1.
Siguiente indice de escritura: 2
pool-1-thread-1 escribio  3 en el elemento 2.
Siguiente indice de escritura: 3
pool-1-thread-2 escribio 11 en el elemento 0.
Siguiente indice de escritura: 4
pool-1-thread-2 escribio 12 en el elemento 4.
Siguiente indice de escritura: 5
pool-1-thread-2 escribio 13 en el elemento 5.
Siguiente indice de escritura: 6

Contenido de ArregloSimple:
11 2 3 0 12 13
```

Primero `pool-1-thread-1` escribió el valor `1` en el elemento `0`. Después `pool-1-thread-2` escribió el valor `11` en el elemento `0`, con lo cual sobrescribió el valor previamente almacenado.

Figura 23.9 | Ejecuta dos objetos `Runnable` para insertar valores en un arreglo compartido. (Parte 2 de 2).

Recuerde que el método `shutdown` de `ExecutorService` regresa de inmediato. Por ende, cualquier código que aparezca después de la llamada al método `shutdown` de `ExecutorService` en la línea 23 seguirá ejecutándose, siempre y cuando el subproceso `main` siga asignado a un procesador. Nos gustaría imprimir el objeto `ArregloSimple` para mostrarle los resultados *después* de que los subprocesos completan sus tareas. Entonces, necesitamos que el programa espere a que los subprocesos se completen antes de que `main` imprima el contenido del objeto `ArregloSimple`. La interfaz `ExecutorService` proporciona el método **`awaitTermination`** para este fin. Este método devuelve el control al que lo llamó, ya sea cuando se completen todas las tareas que se ejecutan

en el objeto `ExecutorService`, o cuando se agote el tiempo de inactividad especificado. Si todas las tareas se completan antes de que se agote el tiempo de `awaitTermination`, este método devuelve `true`; en caso contrario devuelve `false`. Los dos argumentos para `awaitTermination` representan un valor de límite de tiempo y una unidad de medida especificada con una constante de la clase `TimeUnit` (en este caso, `TimeUnit.MINUTES`). En este ejemplo, si ambas tareas se completan antes de que se agote el tiempo de `awaitTermination`, en la línea 32 se muestra el contenido del objeto `ArregloSimple`. En caso contrario, en las líneas 34 y 35 se imprime un mensaje indicando que las tareas no terminaron de ejecutarse antes de que se agotara el tiempo de `awaitTermination`.

Los resultados en la figura 23.9 demuestran los problemas (resaltados en la salida) que se pueden producir debido a la falla en la sincronización del acceso a los datos compartidos. El valor `1` se escribió para el elemento `0`, y más adelante lo sobrescribió el valor `11`. Además, cuando `indiceEscritura` se incrementó a 3, no se escribió nada en ese elemento, como lo indica el `0` en ese elemento del arreglo impreso.

Recuerde que hemos agregado llamadas al método `sleep` de `Thread` entre las operaciones con los datos compartidos para enfatizar la imprevisibilidad de la programación de subprocesos, y para incrementar la probabilidad de producir una salida errónea. Es importante observar que, aun si estas operaciones pudieran proceder a su ritmo normal, de todas formas veríamos errores en la salida del programa. Sin embargo, los procesadores modernos pueden manejar las operaciones simples del método agregar de `ArregloSimple` con tanta rapidez que tal vez no veríamos los errores ocasionados por los dos subprocesos que ejecutan este método en forma concurrente, aun si probáramos el programa docenas de veces. Uno de los retos de la programación con subprocesamiento múltiple es detectar los errores; pueden ocurrir con tan poca frecuencia que un programa erróneo no producirá resultados incorrectos durante la prueba, creando la ilusión de que el programa es correcto.

23.5.2 Cómo compartir datos con sincronización: hacer las operaciones atómicas

Los errores de la salida de la figura 23.9 pueden atribuirse al hecho de que el objeto compartido (`ArregloSimple`) no es **seguro para los subprocesos**; `ArregloSimple` es susceptible a errores si varios subprocesos lo utilizan en forma concurrente. El problema recae en el método `agregar`, el cual almacena el valor de `indiceEscritura`, coloca un nuevo valor en ese elemento y después incrementa a `indiceEscritura`. Dicho método no presentaría problemas en un programa con un solo subproceso. No obstante, si un subproceso obtiene el valor de `indiceEscritura`, no hay garantía de que otro subproceso no pueda llegar e incrementar `indiceEscritura` antes de que el primer subproceso haya tenido la oportunidad de colocar un valor en el arreglo. Si esto ocurre, el primer subproceso estará escribiendo en el arreglo, con base en un valor pasado de `indiceEscritua`; un valor que ya no sea válido. Otra posibilidad es que un subproceso podría **obtener el valor** de `indiceEscritura` después de que otro subproceso agregue un elemento al arreglo, pero antes de que se incremente `indiceEscritura`. En este caso también, el primer subproceso escribiría en el arreglo con base en un valor inválido para `indiceEscritura`.

`ArregloSimple` no es seguro para los subprocesos, ya que permite que cualquier número de subprocesos lean y modifiquen los datos en forma concurrente, lo cual puede producir errores. Para que `ArregloSimple` sea seguro para los subprocesos, debemos asegurar que no haya dos subprocesos que puedan acceder a este objeto al mismo tiempo. También debemos asegurar que mientras un subproceso se encuentre almacenando `indiceEscritura`, agregando un valor al arreglo e incrementando `indiceEscritura`, ningún otro subproceso pueda leer o cambiar el valor de `indiceEscritura`, o modificar el contenido del arreglo en ningún punto durante estas tres operaciones. En otras palabras, deseamos que estas tres operaciones (almacenar `indiceEscritura`, escribir en el arreglo, incrementar `indiceEscritura`) sean una **operación atómica**, la cual no puede dividirse en suboperaciones más pequeñas. Aunque ningún procesador puede llevar a cabo las tres etapas del método agregar en un solo ciclo de reloj para que la operación sea verdaderamente atómica, podemos simular la atomicidad al asegurar que sólo un subproceso lleve a cabo las tres operaciones al mismo tiempo. Cualquier otro subproceso que necesite realizar la operación deberá esperar hasta que el primer subproceso haya terminado la operación `agregar` en su totalidad.

La atomicidad se puede lograr mediante el uso de la palabra clave `synchronized`. Al colocar nuestras tres suboperaciones en una instrucción `synchronized` o en un método `synchronized`, sólo un subproceso a la vez podrá adquirir el bloqueo y realizar las operaciones. Cuando ese subproceso haya completado todas las operaciones en el bloque `synchronized` y libere el bloqueo, otro subproceso podrá adquirir el bloqueo y empezar a ejecutar las operaciones. Esto asegura que un subproceso que ejecuta las operaciones pueda ver los valores actuales de los datos compartidos, y que estos valores no puedan cambiar de manera inesperada, en medio de las operaciones y como resultado de que otro subproceso los modifique.

Observación de ingeniería de software 23.1

Coloque todos los accesos para los datos mutables que puedan compartir varios subprocesos dentro de instrucciones `synchronized`*, o dentro de métodos* `synchronized` *que puedan sincronizarse con el mismo bloqueo. Al realizar varias operaciones con datos compartidos, mantenga el bloqueo durante toda la operación para asegurar que ésta sea, en efecto, atómica.*

En la figura 23.10 se muestra la clase `ArregloSimple` con la sincronización apropiada. Observe que es idéntica a la clase `ArregloSimple` de la figura 23.7, excepto que agregar es ahora un método `synchronized` (línea 19). Por lo tanto, sólo un subproceso a la vez puede ejecutar este método. Reutilizaremos las clases `EscritorArreglo` (figura 23.8) y `PruebaArregloCompartido` (figura 23.9) del ejemplo anterior.

```
1  // Fig. 23.10: ArregloSimple.java
2  // Clase que administra un arreglo simple para compartirlo entre varios subprocesos.
3  import java.util.Random;
4
5  public class ArregloSimple
6  {
7     private final int arreglo[]; // el arreglo entero compartido
8     private int indiceEscritura = 0; // índice del siguiente elemento a escribir
9     private final static Random generador = new Random();
10
11    // construye un objeto ArregloSimple de un tamaño dado
12    public ArregloSimple( int tamanio )
13    {
14       arreglo = new int[ tamanio ];
15    } // fin del constructor
16
17    // agrega un valor al arreglo compartido
18    public synchronized void agregar( int valor )
19    {
20       int posicion = indiceEscritura; // almacena el índice de escritura
21
22       try
23       {
24          // pone el subproceso en inactividad de 0 a 499 milisegundos
25          Thread.sleep( generador.nextInt( 500 ) );
26       } // fin de try
27       catch ( InterruptedException ex )
28       {
29          ex.printStackTrace();
30       } // fin de catch
31
32       // coloca el valor en el elemento apropiado
33       arreglo[ posicion ] = valor;
34       System.out.printf( "%s escribio %2d en el elemento %d.\n",
35          Thread.currentThread().getName(), valor, posicion );
36
37       ++indiceEscritura; // incrementa el índice del siguiente elemento a escribir
38       System.out.printf( "Siguiente indice de escritura: %d\n", indiceEscritura );
39    } // fin del método agregar
40
41    // se utiliza para imprimir el contenido del arreglo entero compartido
42    public String toString()
43    {
44       String cadenaArreglo = "\nContenido de ArregloSimple:\n";
```

Figura 23.10 | Clase que administra un arreglo simple para compartirlo entre varios subprocesos con sincronización. (Parte 1 de 2).

```
45
46         for ( int i = 0; i < arreglo.length; i++ )
47            cadenaArreglo += arreglo[ i ] + " ";
48
49         return cadenaArreglo;
50      } // fin del método toString
51   } // fin de la clase ArregloSimple
```

```
pool-1-thread-1 escribio  1 en el elemento 0.
Siguiente indice de escritura: 1
pool-1-thread-2 escribio 11 en el elemento 1.
Siguiente indice de escritura: 2
pool-1-thread-2 escribio 12 en el elemento 2.
Siguiente indice de escritura: 3
pool-1-thread-2 escribio  13 en el elemento 3.
Siguiente indice de escritura: 4
pool-1-thread-1 escribio  2 en el elemento 4.
Siguiente indice de escritura: 5
pool-1-thread-1 escribio 3 en el elemento 5.
Siguiente indice de escritura: 6

Contenido de ArregloSimple:
1 11 12 13 2 3
```

Figura 23.10 | Clase que administra un arreglo simple para compartirlo entre varios subprocesos con sincronización. (Parte 2 de 2).

En la línea 18 se declara el método como `synchronized`, lo cual hace que todas las operaciones en este método se comporten como una sola operación atómica. En la línea 20 se realiza la primera suboperación: almacenar el valor de `indiceEscritura`. En la línea 33 se define la segunda suboperación, escribir un elemento en el elemento que está en el índice `posicion`. En la línea 37 se incrementa `indiceEscritura`. Cuando el método termina de ejecutarse en la línea 39, el subproceso en ejecución libera el bloqueo de `ArregloSimple`, lo cual hace posible que otro subproceso empiece a ejecutar el método agregar.

En el método `add sincronizado` mediante la palabra clave `synchronized`, imprimimos mensajes en la consola, indicando el progreso de los subprocesos a medida que ejecutan este método, además de realizar las operaciones actuales requeridas para insertar un valor en el arreglo. Hacemos esto de manera que los mensajes se impriman en el orden correcto, con lo cual usted podrá ver si el método está sincronizado apropiadamente, al comparar estos resultados con los del ejemplo anterior, sin sincronización. En ejemplos posteriores seguiremos imprimiendo mensajes de bloques `synchronized` para fines demostrativos; sin embargo, las operaciones de E/S *no deben* realizarse comúnmente en bloques `synchronized`, ya que es importante minimizar la cantidad de tiempo que un objeto permanece "bloqueado".

Tip de rendimiento 23.2

Mantenga la duración de las instrucciones `synchronized` lo más corta posible, mientras mantiene la sincronización necesaria. Esto minimiza el tiempo de espera para los subprocesos bloqueados. Evite realizar operaciones de E/S, cálculos extensos y operaciones que no requieran sincronización, manteniendo un bloqueo.

Otra observación en relación con la seguridad de los subprocesos: hemos dicho que es necesario sincronizar el acceso a todos los datos que puedan compartirse entre varios subprocesos. En realidad, esta sincronización es necesaria sólo para los **datos mutables**, o los datos que puedan cambiar durante su tiempo de vida. Si los datos compartidos no van a cambiar en un programa con subprocesamiento múltiple, entonces no es posible que un subproceso vea valores antiguos o incorrectos, como resultado de que otro subproceso manipule esos datos.

Al compartir datos inmutables entre subprocesos, declare los correspondientes campos de datos como `final` para indicar que los valores de las variables no cambiarán una vez que se inicialicen. Esto evita que los datos compartidos se modifiquen por accidente más adelante en un programa, lo cual podría comprometer la seguridad

de los subprocesos. Al etiquetar las referencias a los objetos como `final` se indica que la referencia no cambiará, pero no se garantiza que el objeto en sí sea inmutable; esto depende por completo de las propiedades del objeto. Sin embargo, aún es buena práctica marcar las referencias que no cambiarán como `final`, ya que esto obliga al constructor del objeto a ser atómico; el objeto estará construido por completo con todos sus campos inicializados, antes de que el programa lo utilice.

Buena práctica de programación 23.1

Siempre declare los campos de datos que no espera modificar como `final`. *Las variables primitivas que se declaran como* `final` *pueden compartirse de manera segura entre los subprocesos. La referencia a un objeto que se declara como* `final` *asegura que el objeto al que refiere estará completamente construido e inicializado antes de que el programa lo utilice; además, esto evita que la referencia apunte a otro objeto.*

23.6 Relación productor/consumidor sin sincronización

En una **relación productor/consumidor**, la porción correspondiente al **productor** de una aplicación genera datos y los almacena en un objeto compartido, y la porción correspondiente al **consumidor** de una aplicación lee esos datos del objeto compartido. La relación productor/consumidor separa la tarea de identificar el trabajo que se va a realizar de las tareas involucradas en realizar ese trabajo. Un ejemplo de una relación productor/consumidor común es la **cola de impresión**. Aunque una impresora podría no estar disponible si queremos imprimir de una aplicación (el productor), aún podemos "completar" la tarea de impresión, ya que los datos se colocan temporalmente en el disco hasta que la impresora esté disponible. De manera similar, cuando la impresora (consumidor) está disponible, no tiene que esperar hasta que un usuario desee imprimir. Los trabajos en la cola de impresión pueden imprimirse tan pronto como la impresora esté disponible. Otro ejemplo de la relación productor/consumidor es una aplicación que copia datos en CDs, colocándolos en un búfer de tamaño fijo, el cual se vacía a medida que la unidad de CD-RW "quema" los datos en el CD.

En una relación productor/consumidor con subprocesamiento múltiple, un **subproceso productor** genera los datos y los coloca en un objeto compartido, llamado **búfer**. Un **subproceso consumidor** lee los datos del búfer. Esta relación requiere sincronización para asegurar que los valores se produzcan y se consuman de manera apropiada. Todas las operaciones con los datos mutables que comparten varios subprocesos (es decir, los datos en el búfer) deben protegerse con un bloqueo para evitar la corrupción, como vimos en la sección 23.5. Las operaciones con los datos del búfer compartidos por un subproceso productor y un subproceso consumidor son también **dependientes del estado**; las operaciones deben proceder sólo si el búfer se encuentra en el estado correcto. Si el búfer se encuentra en un estado en el que no esté completamente lleno, el productor puede producir; si el búfer se encuentra en un estado en el que no esté completamente vacío, el consumidor puede consumir. Todas las operaciones que acceden al búfer deben usar la sincronización para asegurar que los datos se escriban en el búfer, o se lean del búfer, sólo si éste se encuentra en el estado apropiado. Si el productor que trata de colocar los siguientes datos en el búfer determina que éste se encuentra lleno, el subproceso productor debe esperar hasta que haya espacio para escribir un nuevo valor. Si un subproceso consumidor descubre que el búfer está vacío, o que ya se han leído los datos anteriores, también debe esperar a que haya nuevos datos disponibles.

Considere cómo pueden surgir errores lógicos si no sincronizamos el acceso entre varios subprocesos que manipulan datos compartidos. Nuestro siguiente ejemplo (figuras 23.11 a 23.15) implementa una relación productor/consumidor sin la sincronización apropiada. Un subproceso productor escribe los números del 1 al 10 en un búfer compartido: una sola ubicación de memoria compartida entre dos subprocesos (en este ejemplo, una sola variable `int` llamada `bufer` en la línea 6 de la figura 23.14). El subproceso consumidor lee estos datos del búfer compartido y los muestra en pantalla. La salida del programa muestra los valores que el productor escribe (produce) en el búfer compartido, y los valores que el consumidor lee (consume) del búfer compartido.

Cada valor que el subproceso productor escribe en el búfer compartido lo debe consumir exactamente una vez el subproceso consumidor. Sin embargo y por error, los subprocesos en este ejemplo no están sincronizados. Por lo tanto, los datos se pueden perder o desordenar si el productor coloca nuevos datos en el búfer compartido antes de que el consumidor lea los datos anteriores. Además, los datos pueden duplicarse incorrectamente si el consumidor consume datos de nuevo, antes de que el productor produzca el siguiente valor. Para mostrar estas posibilidades, el subproceso consumidor en el siguiente ejemplo mantiene un total de todos los valores que lee. El subproceso productor produce valores del 1 al 10. Si el consumidor lee cada valor producido una, y sólo una vez, el total será de 55. No obstante, si ejecuta este programa varias veces, podrá ver que el total no es siempre 55

(como se muestra en los resultados de la figura 23.10). Para enfatizar este punto, los subprocesos productor y consumidor en el ejemplo quedan inactivos durante intervalos aleatorios de hasta tres segundos, entre la realización de sus tareas. Por ende, no sabemos cuándo el subproceso productor tratará de escribir un nuevo valor, o cuándo el subproceso consumidor tratará de leer uno.

El programa consiste en la interfaz `Bufer` (figura 23.11) y cuatro clases: `Productor` (figura 23.12), `Consumidor` (figura 23.13), `BuferSinSincronizacion` (figura 23.14) y `PruebaBuferCompartido` (figura 23.15). La interfaz `Bufer` declara los métodos `establecer` (línea 6) y `obtener` (línea 9) que un objeto `Bufer` debe implementar para permitir que el subproceso `Productor` coloque un valor en el `Bufer` y el proceso `Consumidor` obtenga un valor del `Bufer`, respectivamente. Algunos programadores prefieren llamar a estos métodos poner (`put`) y tomar (`take`), respectivamente. En posteriores ejemplos, los métodos `establecer` y `obtener` llamarán métodos que lanzan excepciones `InterruptedException`. Aquí declaramos a cada uno de estos métodos con una cláusula throws, para no tener que modificar esta interfaz para los ejemplos posteriores. En la figura 23.14 se muestra la implementación de esta interfaz.

```
1  // Fig. 23.11: Bufer.java
2  // La interfaz Bufer especifica los métodos que el Productor y el Consumidor llaman.
3  public interface Bufer
4  {
5     // coloca valor int value en Bufer
6     public void establecer( int valor ) throws InterruptedException;
7
8     // obtiene valor int de Bufer
9     public int obtener() throws InterruptedException;
10 } // fin de la interfaz Bufer
```

Figura 23.11 | La interfaz Bufer especifica los métodos que el `Productor` y el `Consumidor` llaman.

La clase `Productor` (figura 23.12) implementa a la interfaz `Runnable`, lo cual le permite ejecutarse como una tarea en un subproceso separado. El constructor (líneas 11 a 14) inicializa la referencia `Bufer` llamada `ubicacionCompartida` con un objeto creado en `main` (línea 14 de la figura 23.15), y que se pasa al constructor en el parámetro `compartido`. Como veremos, éste es un objeto `BuferSinSincronizacion` que implementa a la interfaz `Bufer` sin sincronizar el acceso al objeto compartido. El subproceso `Productor` en este programa ejecuta las tareas especificadas en el método `run` (líneas 17 a 39). Cada iteración del ciclo (líneas 21 a 35) invoca al método `sleep` de `Thread` (línea 25) para colocar el subproceso `Productor` en el estado *en espera sincronizado* durante un intervalo de tiempo aleatorio entre 0 y 3 segundos. Cuando el subproceso despierta, en la línea 26 se pasa el valor de la variable de `control` cuenta al método establecer del objeto `Bufer` para `establecer` el valor del búfer compartido. En la línea 27 se mantiene un total de todos los valores producidos hasta ahora, y en la línea 28 se imprime ese valor. Cuando el ciclo termina, en las líneas 37 y 38 se muestra un mensaje indicando que el `Productor` ha dejado de producir datos, y está terminando. A continuación, el método `run` termina, lo cual indica que el `Productor` completó su tarea. Es importante observar que cualquier método que se llama desde el método `run` de `Runnable` (por ejemplo, el método `establecer` de `Bufer`) se ejecuta como parte del subproceso de ejecución de esa tarea. Este hecho se vuelve importante en la sección 23.7, en donde agregamos la sincronización a la relación productor/consumidor.

```
1  // Fig. 23.12: Productor.java
2  // Productor con un método run que inserta los valores del 1 al 10 en el búfer.
3  import java.util.Random;
4
5  public class Productor implements Runnable
6  {
```

Figura 23.12 | `Productor` con un método `run` que inserta los valores del 1 al 10 en el búfer. (Parte 1 de 2).

```
 7     private final static Random generador = new Random();
 8     private final Bufer ubicacionCompartida; // referencia al objeto compartido
 9
10     // constructor
11     public Productor( Bufer compartido )
12     {
13        ubicacionCompartida = compartido;
14     } // fin del constructor de Productor
15
16     // almacena valores del 1 al 10 en ubicacionCompartida
17     public void run()
18     {
19        int suma = 0;
20
21        for ( int cuenta = 1; cuenta <= 10; cuenta++ )
22        {
23           try // permanece inactivo de 0 a 3 segundos, después coloca valor en Bufer
24           {
25              Thread.sleep( generador.nextInt( 3000 ) ); // periodo de inactividad
                aleatorio
26              ubicacionCompartida.establecer( cuenta ); // establece el valor en el búfer
27              suma += cuenta; // incrementa la suma de los valores
28              System.out.printf( "\t%2d\n", suma );
29           } // fin de try
30           // si las líneas 25 o 26 se interrumpen, imprime el rastreo de la pila
31           catch ( InterruptedException excepcion )
32           {
33              excepcion.printStackTrace();
34           } // fin de catch
35        } // fin de for
36
37        System.out.println(
38           "Productor termino de producir\nTerminando Productor" );
39     } // fin del método run
40  } // fin de la clase Productor
```

Figura 23.12 | `Productor` con un método `run` que inserta los valores del 1 al 10 en el búfer. (Parte 2 de 2).

La clase `Consumidor` (figura 23.13) también implementa a la interfaz `Runnable`, lo cual permite al `Consumidor` ejecutarse en forma concurrente con el `Productor`. El constructor (líneas 11 a 14) inicializa la referencia `Bufer` llamada `ubicacionCompartida` con un objeto que implementa a la interfaz `Bufer` creada en `main` (figura 23.15), y se pasa al constructor como el parámetro `compartido`. Como veremos, éste es el mismo objeto `BuferSinSincronizacion` que se utiliza para inicializar el objeto `Productor`; por ende, los dos subprocesos comparten el mismo objeto. El subproceso `Consumidor` en este programa realiza las tareas especificadas en el método `run` (líneas 17 a 39). El ciclo en las líneas 21 a 35 itera 10 veces. Cada iteración invoca al método `sleep` de `Thread` (línea 26) para poner al subproceso `Consumidor` en el estado *en espera sincronizado* durante un tiempo máximo de hasta 3 segundos. A continuación, en la línea 27 se utiliza el método obtener de `Bufer` para `obtener` el valor en el búfer compartido, y después se `suma` el valor a la variable suma. En la línea 28 se muestra el total de todos los valores consumidos hasta ese momento. Cuando el ciclo termina, en las líneas 37 y 38 se muestra una línea indicando la suma de los valores consumidos. Después el método `run` termina, lo cual indica que el `Consumidor` completó su tarea. Una vez que ambos subprocesos entran al estado *terminado*, el programa termina.

[*Nota:* llamamos al método `sleep` en el método `run` de las clases `Productor` y `Consumidor` para enfatizar el hecho de que en las aplicaciones con subprocesamiento múltiple, es impredecible saber cuándo va a realizar su tarea cada subproceso, y por cuánto tiempo realizará la tarea cuando tenga un procesador. Por lo general, estas cuestiones de programación de subprocesos son responsabilidad del sistema operativo de la computadora, lo cual está más allá del control del desarrollador de Java. En este programa, las tareas de nuestro subproceso son bastante simples: el `Productor` escribe los valores 1 a 10 en el búfer, y el `Consumidor` lee 10 valores del búfer y `suma` cada

```
1  // Fig. 23.13: Consumidor.java
2  // Consumidor con un método run que itera y lee 10 valores del búfer.
3  import java.util.Random;
4
5  public class Consumidor implements Runnable
6  {
7     private final static Random generador = new Random();
8     private final Bufer ubicacionCompartida; // referencia al objeto compartido
9
10    // constructor
11    public Consumidor( Bufer compartido )
12    {
13       ubicacionCompartida = compartido;
14    } // fin del constructor de Consumidor
15
16    // lee el valor de ubicacionCompartida 10 veces y suma los valores
17    public void run()
18    {
19       int suma = 0;
20
21       for ( int cuenta = 1; cuenta <= 10; cuenta++ )
22       {
23          // permanece inactivo de 0 a 3 segundos, lee un valor del búfer y lo agrega a
             suma
24          try
25          {
26             Thread.sleep( generador.nextInt( 3000 ) );
27             suma += ubicacionCompartida.obtener();
28             System.out.printf( "t\t\t%2d\n", suma );
29          } // fin de try
30          // si las líneas 26 o 27 se interrumpen, imprime el rastreo de la pila
31          catch ( InterruptedException excepcion )
32          {
33             excepcion.printStackTrace();
34          } // fin de catch
35       } // fin de for
36
37       System.out.printf( "\n%s %d\n%s\n",
38          "Consumidor leyo valores, el total es", suma, "Terminando Consumidor" );
39    } // fin del método run
40 } // fin de la clase Consumidor
```

Figura 23.13 | `Consumidor` con un método `run` que itera y lee 10 valores del búfer.

valor a la variable `suma`. Sin la llamada al método `sleep`, y si el `Productor` se ejecuta primero, dado que hoy en día existen procesadores increíblemente rápidos, es muy probable que el `Productor` complete su tarea antes de que el `Consumidor` tenga oportunidad de ejecutarse. Si el `Consumidor` se ejecutara primero, probablemente consumiría datos basura diez veces y después terminaría antes de que el `Productor` pudiera producir el primer valor real.]

La clase `BuferSinSincronizacion` (figura 23.14) implementa a la interfaz `Bufer` (línea 4). Un objeto de esta clase se comparte entre el `Productor` y el `Consumidor`. En la línea 6 se declara la variable de instancia `bufer` y se inicializa con el valor `-1`. Este valor se utiliza para demostrar el caso en el que el `Consumidor` intenta consumir un valor antes de que el `Productor` coloque siquiera un valor en `bufer`. Los métodos `establecer` (líneas 9 a 13) y `obtener` (líneas 16 a 20) no sincronizan el acceso al campo `bufer`. El método `establecer` simplemente asigna su argumento a `bufer` (línea 12), y el método `obtener` simplemente devuelve el valor de `bufer` (línea 19).

La clase `PruebaBuferCompartido` contiene el método `main` (líneas 9 a 25). En la línea 11 se crea un objeto `ExecutorService` para ejecutar los objetos `Runnable Productor` y `Consumidor`. En la línea 14 se crea un objeto `BuferSinSincronizacion` y se asigna a la variable `Bufer` llamada `ubicacionCompartida`. Este objeto

```
1  // Fig. 23.14: BuferSinSincronizacion.java
2  // BuferSinSincronizacion mantiene el entero compartido que utilizan los
3  // subprocesos productor y consumidor mediante los métodos establecer y obtener.
4  public class BuferSinSincronizacion implements Bufer
5  {
6     private int bufer = -1; // compartido por los subprocesos productor y consumidor
7
8     // coloca el valor en el búfer
9     public void establecer( int valor ) throws InterruptedException
10    {
11       System.out.printf( "Productor escribe\t%2d", valor );
12       bufer = valor;
13    } // fin del método establecer
14
15    // devuelve el valor del búfer
16    public int obtener() throws InterruptedException
17    {
18       System.out.printf( "Consumidor lee\t\t%2d", bufer );
19       return bufer;
20    } // fin del método obtener
21 } // fin de la clase BuferSinSincronizacion
```

Figura 23.14 | `BuferSinSincronizacion` mantiene el entero compartido que utilizan los subprocesos productor y consumidor mediante los métodos `establecer` y `obtener`.

almacena los datos que compartirán los subprocesos `Productor` y `Consumidor`. En las líneas 23 y 24 se crean y ejecutan los objetos `Productor` y `Consumidor`. Observe que los constructores de `Productor` y `Consumidor` reciben el mismo objeto `Bufer` (`ubicacionCompartida`), por lo que cada objeto se inicializa con una referencia al mismo objeto `Bufer`. Estas líneas también inician de manera implícita los subprocesos, y llaman al método `run` de cada objeto `Runnable`. Por último, en la línea 26 se hace una llamada al método `shutdown`, de manera que la aplicación pueda terminar cuando los subprocesos que ejecutan al `Productor` y al `Consumidor` completen sus tareas. Cuando `main` termina (línea 27), el subproceso principal de ejecución entra al estado *terminado*. De acuerdo con las generalidades de este ejemplo, nos gustaría que el `Productor` se ejecutara primero, y que el `Consumidor` consumiera cada valor producido por el `Productor` sólo una vez. Sin embargo, al estudiar los primeros resultados de la figura 23.15, podemos ver que el `Productor` escribe los valores `1`, `2` y `3` antes de que el `Consumidor` lea su primer valor (`3`). Por lo tanto, los valores `1` y `2` se pierden. Más adelante se pierden los valores `5`, `6` y `9`, mientras que `7` y `8` se leen dos veces y `10` se lee cuatro veces. Así, los primeros resultados producen un total incorrecto de 77, en vez del total correcto de 55. En el segundo conjunto de resultados, el `Consumidor` lee el valor `-1` antes de que el `Productor` escriba siquiera un valor. El `Consumidor` lee el valor `1` cinco veces antes de que el `Productor` escriba el valor `2`. Mientras tanto, los valores `5`, `7`, `8`, `9` y `10` se pierden todos; los últimos cuatro debido a que el `Consumidor` termina antes que el `Productor`. Se muestra un total incorrecto del consumidor de 19. (Las líneas en la salida, en donde el `Productor` o el `Consumidor` han actuado fuera de orden, aparecen resaltadas). Este ejemplo demuestra con claridad que el acceso a un objeto compartido por parte de subprocesos concurrentes se debe controlar con cuidado, ya que de no ser así, un programa podría producir resultados incorrectos.

Para resolver los problemas de los datos perdidos y duplicados, en la sección 23.7 se presenta un ejemplo en el que usamos un objeto `ArrayBlockingQueue` (del paquete `java.util.concurrent`) para sincronizar el acceso al objeto compartido, con lo cual se garantiza que cada uno de los valores se procesará una, y sólo una vez.

```
1  // Fig. 23.15: PruebaBuferCompartido.java
2  // Aplicación con dos subprocesos que manipulan un búfer sin sincronización.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
```

Figura 23.15 | Aplicación con dos subprocesos que manipulan un búfer sin sincronización. (Parte 1 de 3).

```
 5
 6  public class PruebaBuferCompartido
 7  {
 8     public static void main( String[] args )
 9     {
10        // crea nueva reserva de subprocesos con dos subprocesos
11        ExecutorService aplicacion = Executors.newCachedThreadPool();
12
13        // crea objeto BuferSinSincronizacion para almacenar valores int
14        Bufer ubicacionCompartida = new BuferSinSincronizacion();
15
16        System.out.println(
17           "Accion\t\t\tValor\tSuma producidos\tSuma consumidos" );
18        System.out.println(
19           "------\t\t\t-----\t---------------\t---------------\n" );
20
21        // ejecuta el Productor y el Consumidor; a cada uno de ellos
22        // le proporciona acceso a ubicacionCompartida
23        aplicacion.execute( new Productor( ubicacionCompartida ) );
24        aplicacion.execute( new Consumidor( ubicacionCompartida ) );
25
26        aplicacion.shutdown(); // termina la aplicación cuando se completan las tareas
27     } // fin de main
28  } // fin de la clase PruebaBuferCompartido
```

```
Accion                  Valor   Suma producidos   Suma consumidos
------                  -----   ---------------   ---------------

Productor escribe        1       1
Productor escribe        2       3                ___ 1  se pierde
Productor escribe        3       6                ___ 2  se pierde
Consumidor lee           3                      3
Productor escribe        4      10
Consumidor lee           4                      7
Productor escribe        5      15
Productor escribe        6      21                ___ 5  se pierde
Productor escribe        7      28                ___ 5  se pierde
Consumidor lee           7                     14
Consumidor lee           7                     21 ___ 7  se lee de nuevo
Productor escribe        8      36
Consumidor lee           8                     29
Consumidor lee           8                     37 ___ 8  se lee de nuevo
Productor escribe        9      45
Productor escribe       10      55                ___ 9  se pierde

Productor termino de producir
Terminando Productor
Consumidor lee          10                     47
Consumidor lee          10                     57 ___ 10 se lee de nuevo
Consumidor lee          10                     67 ___ 10 se lee de nuevo
Consumidor lee          10                     77 ___ 10 se lee de nuevo

Consumidor leyo valores, el total es 77
Terminando Consumidor
```

Figura 23.15 | Aplicación con dos subprocesos que manipulan un búfer sin sincronización. (Parte 2 de 3).

```
Accion                  Valor   Suma producidos   Suma consumidos
------                  -----   ---------------   ---------------
Consumidor lee           -1                      -1 ___ lee -1 datos incorrectos
Productor escribe         1       1
Consumidor lee            1                       0
Consumidor lee            1                       1 ___ 1  se lee de nuevo
Consumidor lee            1                       2 ___ 1  se lee de nuevo
Consumidor lee            1                       3 ___ 1  se lee de nuevo
Consumidor lee            1                       4 ___ 1  se lee de nuevo
Productor escribe         2       3
Consumidor lee            2                       6
Productor escribe         3       6
Consumidor lee            3                       9
Productor escribe         4      10
Consumidor lee            4                      13
Productor escribe         5      15
Productor escribe         6      21                  ___ 5  se pierde
Consumidor lee            6                      19

Consumidor leyo valores, el total es 19
Terminando Consumidor
Productor escribe         7      28                  ___ 7  nunca se lee
Productor escribe         8      36                  ___ 8  nunca se lee
Productor escribe         9      45                  ___ 9  nunca se lee
Productor escribe        10      55                  ___ 10 nunca se lee

Productor termino de producir
Terminando Productor
```

Figura 23.15 | Aplicación con dos subprocesos que manipulan un búfer sin sincronización. (Parte 3 de 3).

23.7 Relación productor/consumidor: `ArrayBlockingQueue`

Una manera de sincronizar los subprocesos productor y consumidor es utilizar las clases del paquete de concurrencia de Java, el cual encapsula la sincronización por nosotros. Java incluye la clase **`ArrayBlockingQueue`** (del paquete `java.util.concurrent`); una clase de búfer completamente implementada, segura para los subprocesos, que implementa a la interfaz **`BlockingQueue`**. Esta interfaz extiende a la interfaz `Queue` que vimos en el capítulo 19 y declara los métodos **`put`** y **`take`**, los equivalentes con bloqueo de los métodos `offer` y `poll` de `Queue`, respectivamente. El método `put` coloca un elemento al final del objeto `BlockingQueue`, y espera si la cola está llena. El método `take` elimina un elemento de la parte inicial del objeto `BlockingQueue`, y espera si la cola está vacía. Estos métodos hacen que la clase `ArrayBlockingQueue` sea una buena opción para implementar un búfer compartido. Debido a que el método `put` bloquea hasta que haya espacio en el búfer para escribir datos, y el método `take` bloquea hasta que haya nuevos datos para leer, el productor debe producir primero un valor, el consumidor sólo consume correctamente hasta después de que el productor escribe un valor, y el productor produce correctamente el siguiente valor (después del primero) sólo hasta que el consumidor lea el valor anterior (o primero). `ArrayBlockingQueue` almacena los datos compartidos en un arreglo. El tamaño de este arreglo se especifica como argumento para el constructor de `ArrayBlockingQueue`. Una vez creado, un objeto `ArrayBlockingQueue` tiene su tamaño fijo y no se expandirá para dar cabida a más elementos.

El programa de las figuras 23.16 y 23.17 demuestra a un `Productor` y un `Consumidor` accediendo a un objeto `ArrayBlockingQueue`. La clase `BuferBloqueo` (figura 23.16) utiliza un objeto `ArrayBlockingQueue` que almacena un objeto `Integer` (línea 7). En la línea 11 se crea el objeto `ArrayBlockingQueue` y se pasa `1` al constructor, para que el objeto contenga un solo valor, como hicimos con el objeto `BuferSinSincronizacion` de la figura 23.14. Observe que en las líneas 7 y 11 utilizamos genéricos, los cuales se describen en los capítulos 18 y 19. En la sección 23.9 hablaremos sobre los búferes con varios elementos. Debido a que nuestra clase `BuferBloqueo` utiliza la clase `ArrayBlockingQueue` (segura para los subprocesos) para administrar el objeto al búfer compartido, `BuferBloqueo` es en sí segura para los subprocesos, aun cuando no hemos implementado la sincronización nosotros mismos.

```
1  // Fig. 23.16: BuferBloqueo.java
2  // Crea un búfer sincronizado, usando la clase ArrayBlockingQueue.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BuferBloqueo implements Bufer
6  {
7     private final ArrayBlockingQueue<Integer> bufer; // bufer compartido
8
9     public BuferBloqueo()
10    {
11       bufer = new ArrayBlockingQueue<Integer>( 1 );
12    } // fin del constructor de BuferBloqueo
13
14    // coloca un valor en el búfer
15    public void establecer( int valor ) throws InterruptedException
16    {
17       bufer.put( valor ); // coloca el valor en el búfer
18       System.out.printf( "%s%2d\t%s%d\n", "Productor escribe ", valor,
19          "Celdas de Bufer ocupadas: ", bufer.size() );
20    } // fin del método establecer
21
22    // devuelve el valor del búfer
23    public int obtener() throws InterruptedException
24    {
25       int valorLeido = 0; // inicializa el valor leído del búfer
26
27       valorLeido = bufer.take(); // elimina el valor del búfer
28       System.out.printf( "%s  %2d\t%s%d\n", "Consumidor lee ",
29          valorLeido, "Celdas de Bufer ocupadas: ", bufer.size() );
30
31       return valorLeido;
32    } // fin del método obtener
33 } // fin de la clase BuferBloqueo
```

Figura 23.16 | Crea un búfer sincronizado, usando la clase `ArrayBlockingQueue`.

`BuferBloqueo` implementa a la interfaz `Bufer` (figura 23.11) y utiliza las clases `Productor` (figura 23.12 modificada para eliminar la línea 28) y `Consumidor` (figura 23.13 modificada para eliminar la línea 28) del ejemplo en la sección 23.6. Este método demuestra que los subprocesos que acceden al objeto compartido no están conscientes de que los accesos a su búfer están ahora sincronizados. La sincronización se maneja por completo en los métodos `establecer` y `obtener` de `BuferBloqueo`, al llamar a los métodos sincronizados `put` y `take` de `ArrayBlockingQueue`, respectivamente. Así, los objetos `Runnable Productor` y `Consumidor` están sincronizados en forma apropiada, con sólo llamar a los métodos `establecer` y `obtener` del objeto compartido.

En la línea 17, en el método `establecer` (líneas 15 a 20) se hace una llamada al método `put` del objeto `ArrayBlockingQueue`. La llamada a este método realiza un bloqueo (si es necesario) hasta que haya espacio en el `bufer` para colocar el `valor`. El método `obtener` (líneas 23 a 32) llama al método `take` del objeto `ArrayBlockingQueue` (línea 27). La llamada a este método realiza un bloqueo (si es necesario) hasta que haya un elemento en el `bufer` que se pueda eliminar. En las líneas 18 y 19, y en las líneas 28 y 29, se utiliza el método `size` del objeto `ArrayBlockingQueue` para mostrar el número total de elementos que se encuentran actualmente en el objeto `ArrayBlockingQueue`.

La clase `PruebaBuferBloqueo` (figura 23.17) contiene el método `main` que inicia la aplicación. En la línea 11 se crea un objeto `ExecutorService`, y en la línea 14 se crea un objeto `BuferBloqueo` y se asigna su referencia a la variable `Bufer` llamada `ubicacionCompartida`. En las líneas 16 y 17 se ejecutan los objetos `Runnable Productor` y `Consumidor`. En la línea 19 se hace una llamada al método `shutdown` para terminar la aplicación cuando los subprocesos terminen de ejecutar las tareas `Productor` y `Consumidor`.

Aunque los métodos `put` y `take` de `ArrayBlockingQueue` están sincronizados en forma apropiada, los métodos `establecer` y `obtener` de `BuferBloqueo` (figura 23.16) no se declaran como sincronizados. Por ende,

```
1  // Fig 23.17: PruebaBuferBloqueo.java
2  // Muestra a dos subprocesos manipulando un búfer con bloqueo.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class PruebaBuferBloqueo
7  {
8     public static void main( String[] args )
9     {
10       // crea nueva reserva de subprocesos con dos subprocesos
11       ExecutorService aplicacion = Executors.newCachedThreadPool();
12
13       // crea objeto BuferBloqueo para almacenar valores int
14       Bufer ubicacionCompartida = new BuferBloqueo();
15
16       aplicacion.execute( new Productor( ubicacionCompartida ) );
17       aplicacion.execute( new Consumidor( ubicacionCompartida ) );
18
19       aplicacion.shutdown();
20    } // fin de main
21 } // fin de la clase PruebaBuferBloqueo
```

```
Productor escribe   1       Celdas de Bufer ocupadas: 1
Consumidor lee      1       Celdas de Bufer ocupadas: 0
Productor escribe   2       Celdas de Bufer ocupadas: 1
Consumidor lee      2       Celdas de Bufer ocupadas: 0

Productor escribe   3       Celdas de Bufer ocupadas: 1
Consumidor lee      3       Celdas de Bufer ocupadas: 0
Productor escribe   4       Celdas de Bufer ocupadas: 1
Consumidor lee      4       Celdas de Bufer ocupadas: 0
Productor escribe   5       Celdas de Bufer ocupadas: 1
Consumidor lee      5       Celdas de Bufer ocupadas: 0
Productor escribe   6       Celdas de Bufer ocupadas: 1
Consumidor lee      6       Celdas de Bufer ocupadas: 0
Productor escribe   7       Celdas de Bufer ocupadas: 1
Consumidor lee      7       Celdas de Bufer ocupadas: 0
Productor escribe   8       Celdas de Bufer ocupadas: 1
Consumidor lee      8       Celdas de Bufer ocupadas: 0
Productor escribe   9       Celdas de Bufer ocupadas: 1
Consumidor lee      9       Celdas de Bufer ocupadas: 0
Productor escribe  10       Celdas de Bufer ocupadas: 1

Productor termino de producir
Terminando Productor
Consumidor lee     10       Celdas de Bufer ocupadas: 0

Consumidor leyo valores, el total es 55
Terminando Consumidor
```

Figura 23.17 | Muestra a dos subprocesos manipulando un búfer con bloqueo.

las instrucciones que se realizan en el método `establecer` (la operación put en la línea 19, y la salida en las líneas 20 y 21) no son atómicas; tampoco lo son las instrucciones en el método `obtener` (la operación `take` en la línea 36, y la salida en las líneas 37 y 38). Por lo tanto, no hay garantía de que cada operación de salida ocurrirá justo después de la correspondiente operación `put` o `take`, y los resultados pueden aparecer fuera de orden. Aun si lo hacen, el objeto `ArrayBlockingQueue` está sincronizando de manera correcta el acceso a los datos, como se puede ver mediante el hecho de que la suma de los valores que lee el consumidor siempre es correcta.

23.8 Relación productor/consumidor con sincronización

El ejemplo anterior mostró cómo varios subprocesos pueden compartir un búfer de un solo elemento en forma segura para los subprocesos, al utilizar la clase `ArrayBlockingQueue` que encapsula la sincronización necesaria para proteger los datos compartidos. Para fines educativos, ahora le explicaremos cómo puede implementar usted mismo un búfer compartido, usando la palabra clave `synchronized`. El uso de un objeto `ArrayBlockingQueue` producirá código con mejor capacidad de mantenimiento y más rendimiento.

El primer paso para sincronizar el acceso al búfer es implementar los métodos `establecer` y `obtener` como métodos `synchronized`. Esto requiere que un subproceso obtenga el bloqueo de monitor en el objeto `Bufer` antes de tratar de acceder a los datos del búfer, pero no resuelve el problema de dependencia asociado con las relaciones productor/consumidor. Debemos asegurar que los subprocesos procedan con una operación sólo si el búfer se encuentra en el estado apropiado. Necesitamos una manera de permitir a nuestros subprocesos esperar, dependiendo de la validez de ciertas condiciones. En el caso de colocar un nuevo elemento en el búfer, la condición que permite que la operación continúe es que el búfer no esté lleno. En el caso de obtener un elemento del búfer, la condición que permite que la operación continúe es que el búfer no esté vacío. Si la condición en cuestión es verdadera, la operación puede continuar; si es falsa, el subproceso debe esperar hasta que la condición se vuelva verdadera. Cuando un subproceso espera en base a una condición, se elimina de la contención para el procesador, se coloca en la cola de espera del objeto y se libera el bloqueo que contiene.

Los métodos `wait`, `notify` y `notifyAll`

Los métodos `wait`, `notify` y `notifyAll`, que se declaran en la clase `Object` y son heredados por las demás clases, se pueden usar con condiciones para hacer que los subprocesos esperen cuando no pueden realizar sus tareas. Si un subproceso obtiene el bloqueo de monitor en un objeto, y después determina que no puede continuar con su tarea en ese objeto sino hasta que se cumpla cierta condición, el subproceso puede llamar al método `wait` de `Object`; esto libera el bloqueo de monitor en el objeto, y el subproceso queda en el estado *en espera* mientras el otro subproceso trata de entrar a la(s) instrucción(es) o método(s) `synchronized` del objeto. Cuando un subproceso que ejecuta una instrucción (o método) `synchronized` completa o cumple con la condición en la que otro subproceso puede estar esperando, puede llamar al método `notify` de `Object` para permitir que un subproceso en espera cambie al estado *ejecutable* de nuevo. En este punto, el subproceso que cambió del estado *en espera* al estado *ejecutable* puede tratar de readquirir el bloqueo de monitor en el objeto. Aun si el subproceso puede readquirir el bloqueo de monitor, tal vez no pueda todavía realizar su tarea en este momento; en ese caso, el subproceso volverá a entrar al estado *en espera* y liberará de manera implícita el bloqueo de monitor. Si un subproceso llama a `notifyAll`, entonces todos los subprocesos que esperan el bloqueo de monitor serán candidatos para readquirir el bloqueo (es decir, todos cambian al estado *ejecutable*). Recuerde que sólo un subproceso a la vez puede adquirir el bloqueo de monitor en el objeto; los demás subprocesos que traten de adquirir el mismo bloqueo de monitor estarán *bloqueados* hasta que el bloqueo de monitor esté disponible de nuevo (es decir, hasta que ningún otro subproceso se esté ejecutando en una instrucción `synchronized` en ese objeto).

Error común de programación 23.1

Es un error si un subproceso llama a `wait`, `notify` o `notifyAll` en un objeto sin haber adquirido un bloqueo para él. Esto produce una excepción `IllegalMonitorStateException`.

Tip para prevenir errores 23.1

Es una buena práctica utilizar a `notifyAll` para notificar a los subprocesos en espera para que cambien al estado ejecutable. Al hacer esto, evitamos la posibilidad de que el programa se olvide de los subprocesos en espera, que de otra forma quedarían aplazados indefinidamente (inanición).

La aplicación de las figuras 23.18 y 23.19 demuestra cómo un `Productor` y un `Consumidor` acceden a un búfer compartido con sincronización. En este caso, el `Productor` siempre produce un valor primero, el `Consumidor` consume correctamente sólo hasta después de que el `Productor` produzca un valor, y el `Productor` produce correctamente el siguiente valor sólo hasta después de que el `Consumidor` consuma el valor anterior (o el primero). Reutilizamos la interfaz `Bufer` y las clases `Productor` y `Consumidor` del ejemplo de la sección 23.6. La sincronización se maneja en los métodos `establecer` y `obtener` de la clase `BuferSincronizado` (figura 23.18), la cual implementa a la interfaz `Bufer` (línea 4). Por ende, los métodos `Productor` y `Consumidor` simplemente llaman a los métodos `synchronized establecer` y `obtener` del objeto compartido.

```
1  // Fig. 23.18: BuferSincronizado.java
2  // Sincronización del acceso a datos compartidos, usando los
3  // métodos wait y notify de Object.
4  public class BuferSincronizado implements Bufer
5  {
6     private int bufer = -1; // compartido por los subprocesos productor y consumidor
7     private boolean ocupado = false; // indica si el búfer está ocupado o no
8
9     // coloca el valor en el búfer
10    public synchronized void establecer( int valor ) throws InterruptedException
11    {
12       // mientras no haya ubicaciones vacías, coloca el subproceso en espera
13       while ( ocupado )
14       {
15          // imprime información del subproceso e información del búfer, después espera
16          System.out.println( "Productor trata de escribir." );
17          mostrarEstado( "Bufer lleno. Productor espera." );
18          wait();
19       } // fin de while
20
21       bufer = valor; // establece el nuevo valor del búfer
22
23       // indica que el productor no puede almacenar otro valor
24       // hasta que el consumidor obtenga el valor actual del búfer
25       ocupado = true;
26
27       mostrarEstado( "Productor escribe " + bufer );
28
29       notifyAll(); // indica al (los) subproceso(s) en espera que entren al estado
         runnable
30    } // fin del método establecer; libera el bloqueo sobre BuferSincronizado
31
32    // devuelve el valor del búfer
33    public synchronized int obtener() throws InterruptedException
34    {
35       // mientras no haya datos para leer, coloca el subproceso en el estado en espera
36       while ( !ocupado )
37       {
38          // imprime la información del subproceso y la información del búfer, después
             espera
39          System.out.println( "Consumidor trata de leer." );
40          mostrarEstado( "Bufer vacio. Consumidor espera." );
41          wait();
42       } // fin de while
43
44       // indica que el productor puede almacenar otro valor
45       // debido a que el consumidor acaba de obtener el valor del búfer
46       ocupado = false;
47
48       mostrarEstado( "Consumidor lee " + bufer );
49
50       notifyAll(); // indica al (los) subproceso(s) en espera que entren al estado
         runnable
51
52       return bufer;
53    } // fin del método obtener; libera el bloqueo sobre BuferSincronizado
54
```

Figura 23.18 | Sincronización del acceso a datos compartidos, usando los métodos `wait` y `notify` de `Object`. (Parte 1 de 2).

```
55      // muestra la operación actual y el estado del búfer
56      public void mostrarEstado( String operacion )
57      {
58         System.out.printf( "40s%d\t\t%b\n\n", operacion, bufer,
59            ocupado );
60      } // fin del método mostrarEstado
61   } // fin de la clase BuferSincronizado
```

Figura 23.18 | Sincronización del acceso a datos compartidos, usando los métodos `wait` y `notify` de `Object`. (Parte 2 de 2).

Campos y métodos de la clase `BuferSincronizado`

La clase `BuferSincronizado` contiene dos campos: `bufer` (línea 6) y `ocupado` (línea 7). Los métodos `establecer` (líneas 10 a 30) y `obtener` (líneas 33 a 53) se declaran como `synchronized`; sólo un subproceso puede llamar a uno de estos métodos a la vez, en un objeto `BuferSincronizado` específico. El campo `ocupado` se utiliza para determinar si es turno del `Productor` o del `Consumidor` para realizar una tarea. Este campo se utiliza en expresiones condicionales en los métodos `establecer` y `obtener`. Si `ocupado` es `false`, el `bufer` está vacío y el `Consumidor` no puede leer el valor de `bufer`, pero el `Productor` puede colocar un valor en este `bufer`. Si `ocupado` es `true`, el `Consumidor` puede leer un valor de `bufer`, pero el `Productor` no puede colocar un valor en este `bufer`.

El método `establecer` *y el subproceso* `Productor`

Cuando el método `run` del subproceso `Productor` invoca al método `establecer` `sincronizado`, el subproceso intenta de manera implícita adquirir el bloqueo de monitor del objeto `BuferSincronizado`. Si el bloqueo de monitor está disponible, el subproceso `Productor` adquiere el bloqueo de manera implícita. Después, el ciclo en las líneas 13 a 19 primero determina si `ocupado` es `true`. Si es así, `bufer` está lleno, por lo que en la línea 16 se imprime un mensaje indicando que el subproceso `Productor` está tratando de escribir un valor, y en la línea 17 se invoca al método `mostrarEstado` (líneas 56 a 60) para imprimir otro mensaje, indicando que `bufer` está lleno y que el subproceso `Productor` está esperando hasta que haya espacio. En la línea 18 se invoca al método `wait` (heredado de `Object` por `BuferSincronizado`) para colocar el subproceso que llamó al método `establecer` (es decir, el subproceso `Productor`) en el estado *en espera* para el objeto `BuferSincronizado`. La llamada a `wait` hace que el subproceso que llama libere implícitamente el bloqueo sobre el objeto `BuferSincronizado`. Esto es importante, ya que el subproceso no puede actualmente realizar su tarea, y porque se debe permitir a otros subprocesos (en este caso, el `Consumidor`) acceder al objeto para permitir que cambie la condición (`ocupado`). Ahora, otro subproceso puede tratar de adquirir el bloqueo del objeto `BuferSincronizado` e invocar al método `establecer` u `obtener` del objeto.

El subproceso `Productor` permanece en el estado *en espera* hasta que otro subproceso notifica al `Productor` que puede continuar; en este punto el `Productor` regresa al estado *ejecutable* y trata de readquirir en forma implícita el bloqueo sobre el objeto `BuferSincronizado`. Si el bloqueo está disponible, el subproceso `Productor` readquiere el bloqueo, y el método `establecer` continúa ejecutándose con la siguiente instrucción después de la llamada a `wait`. Como `wait` se llama en un ciclo, la condición de continuación del ciclo se evalúa de nuevo para determinar si el subproceso puede proceder. Si no es así, entonces `wait` se invoca de nuevo; en caso contrario, el método `establecer` continúa con la siguiente instrucción después del ciclo.

En la línea 21, en el método `establecer` se asigna el valor al `bufer`. En la línea 25 se establece `ocupado` en `true` para indicar que el `bufer` ahora contiene un valor (es decir, un consumidor puede leer el valor, pero un `Productor` no puede colocar todavía un valor ahí). En la línea 27 se invoca al método `mostrarEstado` para imprimir un mensaje que indique que el `Productor` está escribiendo un nuevo valor en el `bufer`. En la línea 29 se invoca el método `notifyAll` (heredado de `Object`). Si hay subprocesos en espera del bloqueo de monitor del objeto `BuferSincronizado`, esos subprocesos entran al estado *ejecutable* y pueden ahora tratar de readquirir el bloqueo. El método `notifyAll` regresa de inmediato, y el método `establecer` regresa entonces al método que hizo la llamada (es decir, el método `run` de `Productor`). Cuando el método `establecer` regresa, libera de manera implícita el bloqueo de monitor sobre el objeto `BuferSincronizado`.

El método `obtener` *y el subproceso* `Consumidor`

Los métodos `obtener` y `establecer` se implementan de manera similar. Cuando el método `run` del subproceso `Consumidor` invoca al método `obtener sincronizado`, el subproceso trata de adquirir el bloqueo de monitor sobre el objeto `BuferSincronizado`. Si el bloqueo está disponible, el subproceso `Consumidor` lo adquiere. Después, el ciclo while en las líneas 36 a 42 determina si `ocupado` es `false`. Si es así, el búfer está vacío, por lo que en la línea 39 se imprime un mensaje indicando que el subproceso `Consumidor` está tratando de leer un valor, y en la línea 40 se invoca el método `mostrarEstado` para imprimir un mensaje que indique que el búfer está vacío, y que el subproceso `Consumidor` está esperando. En la línea 41 se invoca el método `wait` para colocar el subproceso que llamó al método `obtener` (es decir, el `Consumidor`) en el estado *en espera* del objeto `BuferSincronizado`. De nuevo, la llamada a `wait` hace que el subproceso que hizo la llamada libere de manera implícita el bloqueo sobre el objeto `BuferSincronizado`, para que otro subproceso pueda tratar de adquirir el bloqueo del objeto `BuferSincronizado` e invocar al método establecer u obtener del objeto. Si el bloqueo sobre el objeto `BuferSincronizado` no está disponible (por ejemplo, si el `Productor` no ha regresado todavía del método `establecer`), el `Consumidor` se detiene hasta que el bloqueo esté disponible.

El subproceso `Consumidor` permanece en el estado *en espera* hasta que otro subproceso le notifica que puede continuar; en este punto, el subproceso `Consumidor` regresa al estado *ejecutable* y trata de readquirir en forma implícita el bloqueo sobre el objeto `BuferSincronizado`. Si el bloqueo está disponible, el `Consumidor` readquiere el bloqueo y el método `obtener` continúa ejecutándose con la siguiente instrucción después de `wait`. Debido a que la llamada a `wait` ocurre dentro de un ciclo, la condición de continuación del ciclo se evalúa de nuevo para determinar si el subproceso puede continuar con su ejecución. Si no es así, `wait` se invoca de nuevo; en caso contrario, el método `obtener` continúa con la siguiente instrucción después del ciclo. En la línea 46 se establece la variable `ocupado` en `false`, para indicar que el `bufer` está ahora vacío (es decir, un `Consumidor` no puede leer el valor, pero un `Productor` puede colocar otro valor en el `bufer`), en la línea 48 se hace una llamada al método `mostrarEstado` para indicar que el consumidor está leyendo y en la línea 50 se invoca el método `notifyAll`. Si hay subprocesos en el estado *en espera* del bloqueo sobre este objeto `BuferSincronizado`, entran al estado *ejecutable* y pueden ahora readquirir el bloqueo. El método `notifyAll` regresa de inmediato, y después el método `obtener` devuelve el valor de `bufer` al método que lo llamó. Cuando el método `obtener` regresa, el bloqueo sobre el objeto `BuferSincronizado` se libera de manera implícita.

Tip para prevenir errores 23.2

Siempre debe invocar al método `wait` *en un ciclo que evalúe la condición en base a la cual la tarea está esperando. Es posible que un subproceso vuelva a entrar al estado* ejecutable *(a través de una espera sincronizada o debido a que otro subproceso hace una llamada a* `notifyAll`*) antes de que se cumpla la condición. Al evaluar la condición de nuevo, nos aseguramos que el subproceso no se ejecute por error, si se le notificó antes.*

Prueba de la clase `BuferSincronizado`

La clase `PruebaBuferCompartido2` (figura 23.19) es similar a la clase `PruebaBuferCompartido` (figura 23.15). `PruebaBuferCompartido2` contiene el método `main` (líneas 8 a 24), el cual inicia la aplicación. En la línea 11 se crea un objeto `ExecutorService` para ejecutar las tareas `Productor` y `Consumidor`. En la línea 14 se crea un objeto `BuferSincronizado` y se asigna su referencia a la variable `Bufer` llamada `ubicacionCompartida`. Este objeto almacena los datos que se compartirán entre el `Productor` y el `Consumidor`. En las líneas 16 y 17 se muestran los encabezados de columna para los resultados. En las líneas 20 y 21 se ejecuta un `Productor` y un `Consumidor`. Por último, en la línea 23 se hace una llamada al método `shutdown` para terminar la aplicación cuando el `Productor` y el `Consumidor` completen sus tareas. Cuando el método `main` termina (línea 24), el subproceso principal de ejecución termina.

Estudie los resultados de la figura 23.19. Observe que cada entero producido se consume sólo una vez; no se pierden valores, y no se consumen valores más de una vez. La sincronización asegura que el `Productor` produzca un valor sólo cuando el búfer esté vacío, y que el `Consumidor` consuma sólo cuando el búfer esté lleno. El `Productor` siempre va primero, el `Consumidor` espera si el `Productor` no ha producido desde la última vez que el `Consumidor` consumió, y el `Productor` espera si el `Consumidor` no ha consumido todavía el valor que el `Productor` produjo más recientemente. Ejecute este programa varias veces para confirmar que cada entero producido se consuma sólo una vez. En los resultados de ejemplo, observe las líneas resaltadas que indican cuándo deben esperar el `Productor` y el `Consumidor` para realizar sus respectivas tareas.

```
1  // Fig 23.19: PruebaBuferCompartido2.java
2  // La aplicación muestra cómo dos subprocesos manipulan un búfer sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class PruebaBuferCompartido2
7  {
8     public static void main( String[] args )
9     {
10       // crea nueva reserva con dos subprocesos
11       ExecutorService aplicacion = Executors.newCachedThreadPool();
12
13       // crea objeto BuferSincronizado para almacenar valores int
14       Bufer ubicacionCompartida = new BuferSincronizado();
15
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operacion",
17          "Bufer", "Ocupado", "---------", "------\t\t--------" );
18
19       // ejecuta las tareas Productor y Consumidor
20       aplicacion.execute( new Productor( ubicacionCompartida ) );
21       aplicacion.execute( new Consumidor( ubicacionCompartida ) );
22
23       aplicacion.shutdown();
24    } // fin de main
25 } // fin de la clase PruebaBuferCompartido2
```

```
Operacion                               Bufer           Ocupado
---------                               ------          --------

Consumidor trata de leer.
Bufer vacio. Consumidor espera.         -1              false

Productor escribe 1                     1               true

Consumidor lee 1                        1               false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.         1               false

Productor escribe 2                     2               true

Consumidor lee 2                        2               false

Productor escribe 3                     3               true

Consumidor lee 3                        3               false

Productor escribe 4                     4               true

Productor trata de escribir.
Bufer lleno. Productor espera.          4               true

Consumidor lee 4                        4               false

Productor escribe 5                     5               true

Consumidor lee 5                        5               false

Productor escribe 6                     6               true
```

Figura 23.19 | La aplicación muestra cómo dos subprocesos manipulan un búfer sincronizado. (Parte 1 de 2).

```
Productor trata de escribir.
Bufer lleno. Productor espera.          6                   true

Consumidor lee 6                        6                   false

Productor escribe 7                     7                   true

Productor trata de escribir.
Bufer lleno. Productor espera.          7                   true

Consumidor lee 7                        7                   false

Productor escribe 8                     8                   true

Consumidor lee 8                        8                   false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.         8                   false

Productor escribe 9                     9                   true

Consumidor lee 9                        9                   false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.         9                   false

Productor escribe 10                    10                  true

Consumidor lee 10                       10                  false

Productor termino de producir
Terminando Productor

Consumidor leyo valores, el total es 55
Terminando Consumidor
```

Figura 23.19 | La aplicación muestra cómo dos subprocesos manipulan un búfer sincronizado. (Parte 2 de 2).

23.9 Relación productor/consumidor: búferes delimitados

El programa de la sección 23.8 utiliza la sincronización de subprocesos para garantizar que dos subprocesos manipulen correctamente los datos en un búfer compartido. Sin embargo, la aplicación tal vez no tenga un rendimiento óptimo. Si los dos subprocesos operan a distintas velocidades, uno de ellos invertirá más tiempo (o la mayoría de éste) esperando. Por ejemplo, en el programa de la sección 23.8 compartimos una sola variable entera entre los dos subprocesos. Si el subproceso `Productor` produce valores con más rapidez de la que el `Consumidor` puede consumirlos, entonces el subproceso `Productor` espera al `Consumidor`, ya que no hay más ubicaciones en el búfer en donde se pueda colocar el siguiente valor. De manera similar, si el `Consumidor` consume valores con más rapidez de la que el `Productor` los produce, el `Consumidor` espera hasta que el `Productor` coloque el siguiente valor en el búfer compartido. Aun cuando tenemos subprocesos que operan a las mismas velocidades relativas, algunas veces esos subprocesos pueden "salirse de sincronía" durante un periodo de tiempo, lo cual provoca que uno de ellos tenga que esperar al otro. No podemos hacer suposiciones acerca de las velocidades relativas de los subprocesos concurrentes; las interacciones que ocurren con el sistema operativo, la red, el usuario y otros componentes pueden hacer que los subprocesos operen a distintas velocidades. Cuando esto ocurre, los subprocesos esperan. Cuando los subprocesos esperan de manera excesiva, los programas se vuelven menos eficientes, los programas interactivos se vuelven menos responsivos y las aplicaciones sufren retrasos extensos.

Búferes delimitados

Para minimizar la cantidad de tiempo de espera para los subprocesos que comparten recursos y operan a las mismas velocidades promedio, podemos implementar un **bufer delimitado** que proporcione un número fijo de celdas de búfer en las que el `Productor` pueda colocar valores, y de las cuales el `Consumidor` pueda obtener esos valores. (De hecho, ya hemos realizado esto con la clase `ArrayBlockingQueue` en la sección 23.7). Si el `Productor` produce temporalmente valores con más rapidez de la que el `Consumidor` pueda consumirlos, el `Productor` puede escribir otros valores en el espacio adicional del búfer (si hay disponible). Esta capacidad permite al `Productor` realizar su tarea, aún cuando el `Consumidor` no esté listo para obtener el valor que se esté produciendo en ese momento. De manera similar, si el `Consumidor` consume con más rapidez de la que el `Productor` produce nuevos valores, el `Consumidor` puede leer valores adicionales (si los hay) del búfer. Eso permite al `Consumidor` mantenerse ocupado, aún cuando el `Productor` no esté listo para producir valores adicionales.

Observe que, incluso hasta un búfer delimitado es inapropiado si el `Productor` y el `Consumidor` operan consistentemente a distintas velocidades. Si el `Consumidor` siempre se ejecuta con más rapidez que el `Productor`, entonces basta con tener un búfer con una sola ubicación. Las ubicaciones adicionales simplemente desperdiciarían memoria. Si el `Productor` siempre se ejecuta con más rapidez, sólo un búfer con un número "infinito" de ubicaciones podría absorber la producción adicional. No obstante, si el `Productor` y el `Consumidor` se ejecutan aproximadamente a la misma velocidad promedio, un búfer delimitado ayuda a suavizar los efectos de cualquier aceleración o desaceleración ocasional en la ejecución de cualquiera de los dos subprocesos.

La clave para usar un búfer delimitado con un `Productor` y un `Consumidor` que operan aproximadamente a la misma velocidad es proporcionar al búfer suficientes ubicaciones para que pueda manejar la producción "extra" anticipada. Si, durante un periodo de tiempo, determinamos que el `Productor` con frecuencia produce hasta tres valores más de los que el `Consumidor` puede consumir, podemos proporcionar un búfer de por lo menos tres celdas para manejar la producción adicional. Si hacemos el búfer demasiado pequeño, los subprocesos tendrían que esperar más; si hacemos el búfer demasiado grande, se desperdiciaría memoria.

Tip de rendimiento 23.3

Aun cuando se utilice un búfer delimitado, es posible que un subproceso productor pueda llenar el búfer, lo cual obligaría al productor a esperar hasta que un consumidor consumiera un valor para liberar un elemento en el búfer. De manera similar, si el búfer está vacío en algún momento dado, un subproceso consumidor debe esperar hasta que el productor produzca otro valor. La clave para usar un búfer delimitado es optimizar su tamaño para minimizar la cantidad de tiempo de espera de los subprocesos, sin desperdiciar espacio.

Búferes delimitados que utilizan a `ArrayBlockingQueue`

La manera más simple de implementar un búfer delimitado es utilizar un objeto `ArrayBlockingQueue` para el búfer, de manera que se haga cargo de todos los detalles de la sincronización por nosotros. Para ello, podemos reutilizar el ejemplo de la sección 23.7 y pasar simplemente el tamaño deseado para el búfer delimitado al constructor de `ArrayBlockingQueue`. En vez de repetir nuestro ejemplo anterior con `ArrayBlockingQueue` con un tamaño distinto, vamos a presentar un ejemplo que ilustra cómo podemos construir un búfer delimitado por nuestra cuenta. De nuevo, observe que si utilizamos un objeto `ArrayBlockingQueue`, nuestro código tendrá una mejor capacidad de mantenimiento y un mejor rendimiento.

Implementación de su propio búfer delimitado como un búfer circular

El programa de las figuras 23.20 y 23.21 demuestra cómo un `Productor` y un `Consumidor` acceden a un búfer delimitado con sincronización. Implementamos el búfer delimitado en la clase `BuferCircular` (figura 23.20) como un **búfer circular** que utiliza un arreglo compartido de tres elementos. Un búfer circular escribe los elementos de un arreglo y los lee en orden, empezando en la primera celda y avanzando hacia la última. Cuando un `Productor` o `Consumidor` llega al último elemento, regresa al primero y empieza a escribir o leer, respectivamente, de ahí. En esta versión de la relación productor/consumidor, el `Consumidor` consume un valor sólo cuando el arreglo no está vacío y el `Productor` produce un valor sólo cuando el arreglo no está lleno. Las instrucciones que crearon e iniciaron los objetos subproceso en el método `main` de la clase `PruebaBuferCompartido2` (figura 23.19) ahora aparecen en la clase `PruebaBuferCircular` (figura 23.21).

En la línea 5 se inicializa el arreglo `bufer` como un arreglo de tres elementos que representa el búfer circular. La variable `celdasOcupadas` (línea 7) cuenta el número de elementos en `bufer` que contienen datos a leer.

```
1  // Fig. 23.20: BuferCircular.java
2  // Sincronización del acceso a un búfer delimitado compartido, con tres elementos.
3  public class BuferCircular implements Bufer
4  {
5     private final int[] bufer = { -1, -1, -1 }; // búfer compartido
6
7     private int celdasOcupadas = 0; // número de búferes utilizados
8     private int indiceEscritura = 0; // índice del siguiente elemento a escribir
9     private int indiceLectura = 0; // índice del siguiente elemento a leer
10
11    // coloca un valor en el búfer
12    public synchronized void establecer( int valor ) throws InterruptedException
13    {
14       // imprime información del subproceso y del búfer, después espera;
15       // mientras no haya ubicaciones vacías, coloca el subproceso en estado de espera
16       while ( celdasOcupadas == bufer.length )
17       {
18          System.out.printf( "Bufer está lleno. Productor espera.\n" );
19          wait(); // espera hasta que haya una celda libre en el búfer
20       } // fin de while
21
22       bufer[ indiceEscritura ] = valor; // establece nuevo valor del búfer
23
24       // actualiza índice de escritura circular
25       indiceEscritura = ( indiceEscritura + 1 ) % bufer.length;
26
27       ++celdasOcupadas; // una celda más del búfer está llena
28       mostrarEstado( "Productor escribe " + valor );
29       notifyAll(); // notifica a los subprocesos en espera para que lean del búfer
30    } // fin del método establecer
31
32    // devuelve un valor del búfer
33    public synchronized int obtener() throws InterruptedException
34    {
35       // espera hasta que el búfer tenga datos, después lee el valor;
36       // mientras no haya datos para leer, coloca el subproceso en estado de espera
37       while ( celdasOcupadas == 0 )
38       {
39          System.out.printf( "Bufer esta vacio. Consumidor espera.\n" );
40          wait(); // espera hasta que se llene una celda del búfer
41       } // fin de while
42
43       int valorLeido = bufer[ indiceLectura ]; // lee un valor del búfer
44
45       // actualiza índice de lectura circular
46       indiceLectura = ( indiceLectura + 1 ) % bufer.length;
47
48       --celdasOcupadas; // hay una celda ocupada menos en el búfer
49       mostrarEstado( "Consumidor lee " + valorLeido );
50       notifyAll(); // notifica a los subprocesos en espera que pueden escribir en el
         búfer
51
52       return valorLeido;
53    } // fin del método obtener
54
55    // muestra la operación actual y estado del búfer
56    public void mostrarEstado( String operacion )
57    {
58       // imprime operación y número de celdas ocupadas del búfer
```

Figura 23.20 | Sincronización del acceso a un búfer delimitado compartido, con tres elementos. (Parte 1 de 2).

```
59          System.out.printf( "%s%s%d)\n%s", operacion,
60             " (celdas ocupadas del bufer: ", celdasOcupadas, "celdas bufer: " );
61
62          for ( int valor : bufer )
63             System.out.printf( " %2d ", valor ); // imprime los valores que hay en el
               búfer
64
65          System.out.print( "\n              " );
66
67          for ( int i = 0; i < bufer.length; i++ )
68             System.out.print( "---- " );
69
70          System.out.print( "\n              " );
71
72          for ( int i = 0; i < bufer.length; i++ )
73          {
74             if ( i == indiceEscritura && i == indiceLectura )
75                System.out.print( " WR" ); // indice de escritura y de lectura
76             else if ( i == indiceEscritura )
77                System.out.print( " W   " ); // sólo el índice de escritura
78             else if ( i == indiceLectura )
79                System.out.print( "  R  " ); // sólo el índice de lectura
80             else
81                System.out.print( "     " ); // ningún índice
82          } // fin de for
83
84          System.out.println( "\n" );
85       } // fin del método mostrarEstado
86    } // fin de la clase BuferCircular
```

Figura 23.20 | Sincronización del acceso a un búfer delimitado compartido, con tres elementos. (Parte 2 de 2).

Cuando `buferesOcupados` es `0`, no hay datos en el búfer circular y el `Consumidor` debe esperar; cuando `celdasOcupadas` es `3` (el tamaño del búfer circular), el búfer circular está lleno y el `Productor` debe esperar. La variable `indiceEscritura` (línea 8) indica la siguiente ubicación en la que un `Productor` puede colocar un valor. La variable `indiceLectura` (línea 9) indica la posición a partir de la cual un `Consumidor` puede leer el siguiente valor.

El método `establecer` de `BuferCircular` (líneas 12 a 30) realiza las mismas tareas que en la figura 23.18, con unas cuantas modificaciones. El ciclo en las líneas 16 a 20 determina si el `Productor` debe esperar (es decir, todos los búferes están llenos). De ser así, en la línea 18 se indica que el `Productor` está esperando para realizar su tarea. Después, en la línea 19 se invoca el método `wait`, lo cual hace que el `Productor` libere el bloqueo de `BuferCircular` y espere hasta que haya espacio para escribir un nuevo valor en el búfer. Cuando la ejecución continúa en la línea 22 después del ciclo `while`, el valor escrito por el `Productor` se coloca en el búfer circular, en la ubicación `indiceEscritura`. Después, en la línea 25 se actualiza `indiceEscritura` para la siguiente llamada al método `establecer` de `BuferCircular`. Esta línea es la clave para la "circularidad" del búfer. Cuando `indiceEscritura` se incrementa más allá del final del búfer, la línea lo establece en `0`. En la línea 27 se incrementa `celdasOcupadas`, debido a que ahora hay un valor más en el búfer que el `Consumidor` puede leer. A continuación, en la línea 28 se invoca el método `mostrarEstado` (líneas 56 a 85) para actualizar la salida con el valor producido, el número de búferes ocupados, el contenido de esos búferes y los valores actuales de `indiceEscritura` e `indiceLectura`. En la línea 29 se invoca el método `notifyAll` para cambiar los subprocesos en espera al estado *ejecutable*, de manera que un subproceso `Consumidor` en espera (si hay uno) pueda intentar leer nuevamente un valor del búfer.

El método `obtener` de `BuferCircular` (líneas 33 a 53) también realiza las mismas tareas que hizo en la figura 23.18, con unas cuantas modificaciones menores. El ciclo en las líneas 37 a 41 determina si el `Consumidor` debe esperar (es decir, todas las celdas del búfer están vacías). Si el `Consumidor` debe esperar, en la línea 39 se actualiza la salida para indicar que el `Consumidor` está esperando realizar su tarea. Después, en la línea 40 se

invoca el método `wait`, lo cual hace que el subproceso actual libere el bloqueo sobre el `BuferCircular` y espera hasta que haya datos disponibles para leer. Cuando la ejecución continúa en un momento dado en la línea 43, después de que el `Productor` llama a `notifyAll`, a `valorLeido` se le asigna el valor en la ubicación `indiceLectura` en el búfer circular. Después, en la línea 46 se actualiza `indiceLectura` para la siguiente llamada al método `obtener` de `BuferCircular`. En esta línea y en la línea 25 se implementa la "circularidad" del búfer. En la línea 48 se decrementa `celdasOcupadas`, debido a que ahora hay una posición más en el búfer en la que el subproceso `Productor` puede colocar un valor. En la línea 49 se invoca el método `mostrarEstado` para actualizar la salida con el valor consumido, el número de búferes ocupados, el contenido de los búferes y los valores actuales de `indiceEscritura` e `indiceLectura`. En la línea 50 se invoca el método `notifyAll` para permitir que cualquier subproceso `Productor` que esté esperando escribir en el objeto `BuferCircular` intente escribir de nuevo. Después, en la línea 52 se devuelve el valor consumido al método que hizo la llamada.

El método `mostrarEstado` (líneas 56 a 85) imprime en pantalla el estado de la aplicación. En las líneas 62 y 63 se muestran los valores actuales de las celdas del búfer. En la línea 63 se utiliza el método `printf` con un especificador de formato `"%2d"` para imprimir el contenido de cada búfer con un espacio a la izquierda, si es un solo dígito. En las líneas 70 a 82 se imprimen en pantalla los valores actuales de `indiceEscritura` e `indiceLectura` con las letras `W` y `R`, respectivamente.

Prueba de la clase `BuferCircular`

La clase `PruebaBuferCircular` (figura 23.21) contiene el método main que inicia la aplicación. En la línea 11 se crea el objeto `ExecutorService`, y en la línea 14 se crea un objeto `BuferCircular` y se asigna su referencia a la variable `BuferCircular` llamada `ubicacionCompartida`. En la línea 17 se invoca el método `mostrarEstado` de `BuferCircular` para mostrar el estado inicial del búfer. En las líneas 20 y 21 se ejecutan las tareas `Productor` y `Consumidor`. En la línea 23 se hace una llamada al método `shutdown` para terminar la aplicación cuando los subprocesos completen las tareas `Productor` y `Consumidor`.

Cada vez que el `Productor` escribe un valor o el `Consumidor` lee un valor, el programa imprime en pantalla un mensaje indicando la acción realizada (lectura o escritura), el contenido de `bufer` y la ubicación de `indiceEscritura` e `indiceLectura`. En la salida de la figura 23.21, el `Productor` escribe primero el valor `1`. Después, el búfer contiene el valor `1` en la primera celda y el valor `-1` (el valor predeterminado que utilizamos para fines de mostrar los resultados) en las otras dos celdas. El índice de escritura se actualiza a la segunda celda, mientras que el índice de lectura permanece en la primera celda. A continuación, el `Consumidor` lee `1`. El búfer contiene los mismos valores, pero el índice de lectura se ha actualizado a la segunda celda. Después el `Consumidor` trata de leer otra vez, pero el búfer está vacío y el `Consumidor` se ve obligado a esperar. Observe que sólo una vez en esta ejecución del programa fue necesario que uno de los dos subprocesos esperara.

```
1  // Fig 23.21: PruebaBuferCircular.java
2  // Muestra dos subprocesos que manipulan un búfer circular.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class PruebaBuferCircular
7  {
8     public static void main( String[] args )
9     {
10       // crea nueva reserva con dos subprocesos
11       ExecutorService aplicacion = Executors.newCachedThreadPool();
12
13       // crea objeto BuferCircular para almacenar valores int
14       BuferCircular ubicacionCompartida = new BuferCircular();
15
16       // muestra el estado inicial del objeto BuferCircular
17       ubicacionCompartida.mostrarEstado( "Estado inicial" );
```

Figura 23.21 | La aplicación muestra cómo los subprocesos `Productor` y `Consumidor` manipulan un búfer circular. (Parte 1 de 3).

```
18
19         // ejecuta las tareas Productor y Consumidor
20         aplicacion.execute( new Productor( ubicacionCompartida ) );
21         aplicacion.execute( new Consumidor( ubicacionCompartida ) );
22
23         aplicacion.shutdown();
24      } // fin de main
25   } // fin de la clase PruebaBuferCircular
```

```
Estado inicial (celdas ocupadas del bufer: 0)
celdas bufer:   -1   -1   -1
               ---- ---- ----
                WR

Productor escribe 1 (celdas ocupadas del bufer: 1)
celdas bufer:    1   -1   -1
               ---- ---- ----
                 R   W

Consumidor lee 1 (celdas ocupadas del bufer: 0)
celdas bufer:    1   -1   -1
               ---- ---- ----
                     WR

Bufer esta vacio. Consumidor espera.
Productor escribe 2 (celdas ocupadas del bufer: 1)
celdas bufer:    1    2   -1
               ---- ---- ----
                      R   W

Consumidor lee 2 (celdas ocupadas del bufer: 0)
celdas bufer:    1    2   -1
               ---- ---- ----
                          WR

Productor escribe 3 (celdas ocupadas del bufer: 1)
celdas bufer:    1    2    3
               ---- ---- ----
                W          R

Consumidor lee 3 (celdas ocupadas del bufer: 0)
celdas bufer:    1    2    3
               ---- ---- ----
                WR

Productor escribe 4 (celdas ocupadas del bufer: 1)
celdas bufer:    4    2    3
               ---- ---- ----
                 R   W

Productor escribe 5 (celdas ocupadas del bufer: 2)
celdas bufer:    4    5    3
               ---- ---- ----
                 R        W

Consumidor lee 4 (celdas ocupadas del bufer: 1)
celdas bufer:    4    5    3
               ---- ---- ----
                      R   W
```

Figura 23.21 | La aplicación muestra cómo los subprocesos `Productor` y `Consumidor` manipulan un búfer circular. (Parte 2 de 3).

```
Productor escribe 6 (celdas ocupadas del bufer: 2)
celdas bufer:    4    5    6
               ---- ---- ----
                W     R

Productor escribe 7 (celdas ocupadas del bufer: 3)
celdas bufer:    7    5    6
               ---- ---- ----
                     WR

Consumidor lee 5 (celdas ocupadas del bufer: 2)
celdas bufer:    7    5    6
               ---- ---- ----
                     W     R

Productor escribe 8 (celdas ocupadas del bufer: 3)
celdas bufer:    7    8    6
               ---- ---- ----
                          WR

Consumidor lee 6 (celdas ocupadas del bufer: 2)
celdas bufer:    7    8    6
               ---- ---- ----
                 R         W

Consumidor lee 7 (celdas ocupadas del bufer: 1)
celdas bufer:    7    8    6
               ---- ---- ----
                      R   W

Productor escribe 9 (celdas ocupadas del bufer: 2)
celdas bufer:    7    8    9
               ---- ---- ----
                W     R

Consumidor lee 8 (celdas ocupadas del bufer: 1)
celdas bufer:    7    8    9
               ---- ---- ----
                W          R

Consumidor lee 9 (celdas ocupadas del bufer: 0)
celdas bufer:    7    8    9
               ---- ---- ----
                WR

Productor escribe 10 (celdas ocupadas del bufer: 1)
celdas bufer:   10    8    9
               ---- ---- ----
                 R   W

Productor termino de producir
Terminando Productor
Consumidor lee 10 (celdas ocupadas del bufer: 0)
celdas bufer:   10    8    9
               ---- ---- ----
                     WR

Consumidor leyo valores, el total es 55
Terminando Consumidor
```

Figura 23.21 | La aplicación muestra cómo los subprocesos `Productor` y `Consumidor` manipulan un búfer circular. (Parte 3 de 3).

23.10 Relación productor/consumidor: las interfaces Lock y Condition

Aunque la palabra clave `synchronized` proporciona la mayoría de las necesidades de sincronización de subprocesos, Java cuenta con otras herramientas para ayudar en el desarrollo de programas concurrentes. En esta sección, hablaremos sobre las interfaces `Lock` y `Condition`, que se introdujeron en Java SE 5. Estas interfaces proporcionan a los programadores un control más preciso sobre la sincronización de subprocesos, pero su uso es más complicado.

La interfaz `Lock` y la clase `ReentrantLock`

Cualquier objeto puede contener una referencia a un objeto que implemente a la interfaz `Lock` (del paquete `java.util.concurrent.locks`). Un subproceso llama al método `lock` de `Lock` para adquirir el bloqueo. Una vez que un subproceso obtiene un objeto `Lock`, este objeto no permitirá que otro subproceso obtenga el `Lock` sino hasta que el primer subproceso lo libere (llamando al método **`unlock`** de `Lock`). Si varios subprocesos tratan de llamar al método `lock` en el mismo objeto `Lock` y al mismo tiempo, sólo uno de estos subprocesos puede obtener el bloqueo; todos los demás se colocan en el estado *en espera* de ese bloqueo. Cuando un subproceso llama al método `unlock`, se libera el bloqueo sobre el objeto y un subproceso en espera que intente bloquear el objeto puede continuar.

La clase **`ReentrantLock`** (del paquete `java.util.concurrent.locks`) es una implementación básica de la interfaz `Lock`. El constructor de `ReentrantLock` recibe un argumento `boolean`, el cual especifica si el bloqueo tiene una política de equidad. Si el argumento es `true`, la **política de equidad** de `ReentrantLock` es: "el subproceso con más tiempo de espera adquirirá el bloqueo cuando esté disponible". Dicha política de equidad garantiza que nunca ocurra el aplazamiento indefinido (también conocido como inanición). Si el argumento de la política de equidad se establece en `false`, no hay garantía en cuanto a cuál subproceso en espera adquirirá el bloqueo cuando esté disponible.

Observación de ingeniería de software 23.2

El uso de un objeto `ReentrantLock` con una política de equidad evita el aplazamiento indefinido.

Tip de rendimiento 23.4

El uso de un objeto `ReentrantLock` con una política de equidad puede reducir el rendimiento del programa, de una manera considerable.

Los objetos de condición y la interfaz `Condition`

Si un subproceso que posee un objeto `Lock` determina que no puede continuar con su tarea hasta que se cumpla cierta condición, el subproceso puede esperar en base a un **objeto de condición**. El uso de objetos `Lock` nos permite declarar de manera explícita los objetos de condición sobre los cuales un subproceso tal vez tenga que esperar. Por ejemplo, en la relación productor/consumidor los productores pueden esperar en base a un objeto, y los consumidores en base a otro. Esto no es posible cuando se utiliza la palabra clave `synchronized` y el bloqueo de monitor integrado de un objeto. Los objetos de condición se asocian con un objeto `Lock` específico y se crean mediante una llamada al método **`newCondition`** de Lock, el cual devuelve un objeto que implementa a la interfaz **`Condition`** (del paquete `java.util.concurrent.locks`). Para esperar en base a un objeto de condición, el subproceso puede llamar al método **`await`** de `Condition`. Esto libera de inmediato el objeto `Lock` asociado, y coloca al subproceso en el estado *en espera*, en base a ese objeto `Condition`. Así, otros subprocesos pueden tratar de obtener el objeto `Lock`. Cuando un subproceso *ejecutable* completa una tarea y determina que el subproceso *en espera* puede ahora continuar, el subproceso *ejecutable* puede llamar al método **`signal`** de `Condition` para permitir que un subproceso en el estado *en espera* de ese objeto `Condition` regrese al estado *ejecutable*. En este punto, el subproceso que cambió del estado *en espera* al estado *ejecutable* puede tratar de readquirir el objeto `Lock`. Aun si puede readquirir el objeto `Lock`, tal vez el subproceso no pueda todavía realizar su tarea en este momento; en cuyo caso, el subproceso puede llamar al método `await` de `Condition` para liberar el objeto `Lock` y volver a entrar al estado *en espera*. Si hay varios subprocesos en un estado *en espera* de un objeto `Condition` cuando se hace la llamada a `signal`, la implementación predeterminada de `Condition` indica al subproceso con más tiempo de espera que debe cambiar al estado *ejecutable*. Si un subproceso llama al método **`signalAll`** de `Condition`,

entonces todos los subprocesos que esperan esa condición cambian al estado *ejecutable* y se convierten en candidatos para readquirir el objeto `Lock`. Sólo uno de esos subprocesos puede obtener el bloqueo (`Lock`) sobre el objeto; los demás esperarán hasta que el objeto `Lock` esté disponible otra vez. Si el objeto `Lock` tiene una política de equidad, el subproceso con más tiempo de espera adquiere ese objeto `Lock`. Cuando un subproceso termina con un objeto compartido, debe llamar al método `unlock` para liberar al objeto `Lock`.

Error común de programación 23.2

*El **interbloqueo** (deadlock) ocurre cuando un subproceso en espera (al cual llamaremos subproceso1) no puede continuar, debido a que está esperando (ya sea en forma directa o indirecta) a que otro subproceso (al cual llamaremos subproceso2) continúe, mientras que al mismo tiempo, el subproceso2 no puede continuar debido a que está esperando (ya sea en forma directa o indirecta) a que el subproceso 1 continúe. Los dos subprocesos están en espera uno del otro, por lo que las acciones que permitirían a cada subproceso continuar su ejecución nunca ocurrirán.*

Tip para prevenir errores 23.3

Cuando varios subprocesos manipulan a un objeto compartido mediante el uso de bloqueos, debemos asegurarnos que, si un subproceso llama al método `await` para entrar al estado en espera de un objeto de condición, en algún momento dado un subproceso separado llamará al método `signal` de `Condition` para cambiar el subproceso que espera al objeto de condición de vuelta al estado ejecutable*. Si puede haber varios subprocesos esperando el objeto de condición, un subproceso separado puede llamar al método `signalAll` de `Condition` como garantía para asegurar que todos los subprocesos en espera tengan otra oportunidad para realizar sus tareas. Si esto no se hace, puede ocurrir un aplazamiento indefinido (inanición).*

Error común de programación 23.3

Una excepción `IllegalMonitorStateException` ocurre si un subproceso llama a `await`, `signal` o `signalAll` en base a un objeto de condición, sin haber adquirido el bloqueo para ese objeto de condición.

Comparación entre `Lock` y `Condition`, y la palabra clave `synchronized`

En algunas aplicaciones, el uso de objetos `Lock` y `Condition` puede ser preferible a utilizar la palabra clave `synchronized`. Los objetos `Lock` nos permiten interrumpir a los subprocesos en espera, o especificar un tiempo límite para esperar a adquirir un bloqueo, lo cual no es posible si se utiliza la palabra clave `synchronized`. Además, un objeto `Lock` no está restringido a ser adquirido y liberado en el mismo bloque de código, lo cual es el caso con la palabra clave `synchronized`. Los objetos `Condition` nos permiten especificar varios objetos de condición, en base a los cuales los subprocesos pueden esperar. Por ende, es posible indicar a los subprocesos en espera que un objeto de condición específico es ahora verdadero, llamando a `signal` o `signalAll` en ese objeto `Condition`. Con la palabra clave `synchronized`, no hay forma de indicar de manera explícita la condición en la cual esperan los subprocesos y, por lo tanto, no hay forma de notificar a los subprocesos en espera de una condición específica que pueden continuar, sin también indicarlo a los subprocesos que están en espera de otras condiciones. Hay otras posibles ventajas en cuanto al uso de objetos `Lock` y `Condition`, pero debemos tener en cuenta que, por lo general, es mejor utilizar la palabra clave `synchronized`, a menos que nuestra aplicación requiera capacidades avanzadas de sincronización. El uso de las interfaces `Lock` y `Condition` es propenso a errores; no se garantiza la llamada a `unlock`, mientras que el monitor en una instrucción `synchronized` siempre se liberará cuando la instrucción termine de ejecutarse.

Uso de objetos `Lock` y `Condition` para implementar la `sincronización`

Para ilustrar cómo usar las interfaces `Lock` y `Condition`, ahora vamos a implementar la relación productor/consumidor, utilizando objetos `Lock` y `Condition` para coordinar el acceso a un búfer compartido con un solo elemento (figuras 23.22 y 23.23). En este caso, cada valor producido se consume correctamente sólo una vez.

La clase `BuferSincronizado` (figura 23.22) contiene cinco campos. En la línea 11 se crea un nuevo objeto de tipo `ReentrantLock` y se asigna su referencia a la variable `Lock` llamada `bloqueoAcceso`. El objeto `bloqueoReentrante` se crea sin la política de equidad, ya que en cualquier momento dado, sólo un `Productor` o `Consumidor` estará esperando adquirir el objeto `Lock` en este ejemplo. En las líneas 14 y 15 se crean dos objetos `Condition` mediante el uso del método `newCondition` de `Lock`. El objeto `Condition` llamado `puedeEscribir` contiene una cola para un subproceso `Productor`, el cual espera mientras el búfer esté lleno (es decir, hay datos

en el búfer pero el `Consumidor` no los ha leído aún). Si el búfer está lleno, el `Productor` llama al método `await` en este objeto `Condition`. Cuando el `Consumidor` lee datos de un búfer lleno, llama al método `signal` en este objeto `Condition`. El objeto `Condition puedeLeer` contiene una cola para un `Consumidor` que espera mientras el búfer esté vacío (es decir, no hay datos en el búfer para que el `Consumidor` los lea). Si el búfer está vacío, el `Consumidor` llama al método `await` en este objeto `Condition`. Cuando el `Productor` escribe en el búfer vacío, llama al método `signal` en este objeto `Condition`. La variable `int bufer` (línea 17) contiene los datos compartidos. La variable `boolean ocupado` (línea 18) mantiene el rastro acerca de si el búfer contiene datos en un momento dado (que el `Consumidor` debe leer).

```
1  // Fig. 23.22: BuferSincronizado.java
2  // Sincroniza el acceso a un entero compartido, usando las interfaces
3  // Lock y Condition
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6  import java.util.concurrent.locks.Condition;
7
8  public class BuferSincronizado implements Bufer
9  {
10    // Bloqueo para controlar la sincronización con este búfer
11    private final Lock bloqueoAcceso = new ReentrantLock();
12
13    // condiciones para controlar la lectura y escritura
14    private final Condition puedeEscribir = bloqueoAcceso.newCondition();
15    private final Condition puedeLeer = bloqueoAcceso.newCondition();
16
17    private int bufer = -1; // compartido por los subprocesos productor y consumidor
18    private boolean ocupado = false; // indica si el búfer está ocupado
19
20    // coloca un valor int en el búfer
21    public void establecer( int valor ) throws InterruptedException
22    {
23       bloqueoAcceso.lock(); // bloquea este objeto
24
25       // imprime información del subproceso y del búfer, después espera
26       try
27       {
28          // mientras búfer no esté vacío, coloca el subproceso en espera
29          while ( ocupado )
30          {
31             System.out.println( "Productor trata de escribir.");
32             mostrarEstado( "Bufer lleno. Productor espera." );
33             puedeEscribir.await(); // espera hasta que bufer esté vacío
34          } // fin de while
35
36          bufer = valor; // establece el nuevo valor de búfer
37
38          // indica que el productor no puede almacenar otro valor
39          // hasta que el consumidor obtenga el valor actual del búfer
40          ocupado = true;
41
42          mostrarEstado( "Productor escribe " + búfer );
43
44          // indica al subproceso en espera que lea del búfer
45          puedeLeer.signal();
46       } // fin de try
47       finally
48       {
```

Figura 23.22 | Sincroniza el acceso a un entero compartido, usando las interfaces `Lock` y `Condition`. (Parte 1 de 2).

```
49            bloqueoAcceso.unlock(); // desbloquea este objeto
50         } // fin de finally
51      } // fin del método establecer
52
53      // devuelve el valor del búfer
54      public int obtener() throws InterruptedException
55      {
56         int valorLeido = 0; // inicializa el valor que se leyó del búfer
57         bloqueoAcceso.lock(); // bloquea este objeto
58
59         // imprime información del subproceso y del búfer, después espera
60         try
61         {
62            // mientras no haya datos qué leer, coloca el subproceso en espera
63            while ( !ocupado )
64            {
65               System.out.println( "Consumidor trata de leer." );
66               mostrarEstado( "Bufer vacio. Consumidor espera." );
67               puedeLeer.await(); // espera hasta que bufer esté lleno
68            } // fin de while
69
70            // indica que el productor puede almacenar otro valor
71            // porque el consumidor acaba de obtener el valor del búfer
72            ocupado = false;
73
74            valorLeido = bufer; // obtiene el valor del búfer
75            mostrarEstado( "Consumidor lee " + valorLeido );
76
77            // indica al subproceso que espera a que el búfer esté vacío
78            puedeEscribir.signal();
79         } // fin de try
80         finally
81         {
82            bloqueoAcceso.unlock(); // desbloquea este objeto
83         } // fin de finally
84
85         return valorLeido;
86      } // fin del método obtener
87
88      // muestra la operación actual y el estado del búfer
89      public void mostrarEstado( String operacion )
90      {
91         System.out.printf( "%-40s%d\t\t%b\n\n", operacion, bufer,
92            ocupado );
93      } // fin del método mostrarEstado
94   } // fin de la clase BuferSincronizado
```

Figura 23.22 | Sincroniza el acceso a un entero compartido, usando las interfaces `Lock` y `Condition`. (Parte 2 de 2).

En la línea 23, en el método `establecer` se hace una llamada al método `lock` en el objeto `bloqueoAcceso` de `BuferSincronizado`. Si el bloqueo está disponible (es decir, si ningún otro subproceso ha adquirido este bloqueo), el método `lock` regresa de inmediato (este subproceso ahora posee el bloqueo) y el subproceso continúa. Si el bloqueo no está disponible (es decir, si lo posee otro subproceso), este método espera hasta que el otro subproceso libere el bloqueo. Una vez que se adquiere el bloqueo, se ejecuta el bloque `try` en las líneas 26 a 46. En la línea 29 se evalúa `ocupado` para determinar si `bufer` está lleno. Si es así, en las líneas 31 y 32 se muestra un mensaje indicando que el subproceso va a esperar. En la línea 33 se hace una llamada al método `await` de `Condition` en el objeto de condición `puedeEscribir`, lo cual libera temporalmente el objeto `Lock` de `BuferSincronizado` y espera una señal del `Consumidor`, indicando que el `bufer` está disponible para escritura. Cuando `bufer` está

disponible, el método continúa y escribe en `bufer` (línea 36), establece `ocupado` en `true` (línea 40) y muestra un mensaje indicando que el productor escribió un valor (línea 42). En la línea 45 se hace una llamada al método `signal` de `Condition` en el objeto de condición `puedeLeer`, para notificar al `Consumidor` en espera (si hay uno) que el búfer tiene nuevos datos para leer. En la línea 49 se hace una llamada al método `unlock` desde un bloque `finally` para liberar el bloqueo y permitir que el `Consumidor` continúe.

Tip para prevenir errores 23.4

Coloque las llamadas al método `unlock` de Lock en un bloque `finally`. Si se lanza una excepción, `unlock` debe llamarse de todas formas, o de lo contrario ocurriría un interbloqueo.

En la línea 57 del método `obtener` (líneas 54 a 86) se hace una llamada al método `lock` para adquirir el objeto `Lock`. Este método espera hasta que el objeto `Lock` esté disponible. Una vez que se adquiere este objeto `Lock`, en la línea 63 se evalúa si ocupado es `false`, lo cual indica que el búfer está vacío. Si es así, en la línea 67 se hace una llamada al método `await` en el objeto de condición `puedeLeer`. Recuerde que el método `signal` se llama en la variable `puedeLeer`, en el método `establecer` (línea 45). Cuando se hace una indicación al objeto `Condition`, el método `obtener` continúa. En la línea 72 se establece `ocupado` en `false`, en la línea 74 se almacena el valor de `bufer` en `valorLeido` y en la línea 75 se imprime en pantalla el `valorLeido`. Después, en la línea 78 se hace una indicación al objeto de condición `puedeEscribir`. Esto despertará al `Productor` si es que está esperando a que el búfer esté vacío. En la línea 82 se hace una llamada al método `unlock` desde un bloque `finally` para liberar el bloqueo, y en la línea 85 se devuelve `valorLeido` al método que hizo la llamada.

Error común de programación 23.4

Olvidar hacer una indicación mediante `signal` a un subproceso en espera es un error lógico. El subproceso permanecerá en el estado en espera*, lo cual evitará que pueda continuar. Dicha espera puede producir un aplazamiento indefinido o un interbloqueo.*

La clase `PruebaBuferCompartido2` (figura 23.23) es idéntica a la de la figura 23.19. Estudie el conjunto de resultados de la figura 23.23. Observe que cada entero producido se consume exactamente una vez; no se pierden valores, y no se consumen valores más de una vez. Los objetos `Lock` y `Condition` aseguran que el `Productor` y el `Consumidor` no puedan realizar sus tareas, a menos que sea su turno. El `Productor` debe ir primero, el `Consumidor` debe esperar si el `Productor` no ha producido desde la última vez que el `Consumidor` consumió, y el `Productor` debe esperar si el `Consumidor` no ha consumido aún el valor que el `Productor` produjo más recientemente. Ejecute este programa varias veces para confirmar que cada entero producido sea consumido exactamente una vez. En los resultados de ejemplo observe las líneas resaltadas, las cuales indican cuándo deben esperar el `Productor` y el `Consumidor` para realizar sus respectivas tareas.

```
1  // Fig 23.23: PruebaBuferCompartido2.java
2  // Dos subprocesos que manipulan un búfer sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class PruebaBuferCompartido2
7  {
8     public static void main( String[] args )
9     {
10       // crea nueva reserva con dos subprocesos
11       ExecutorService aplicacion = Executors.newCachedThreadPool();
12
13       // crea BuferSincronizado para almacenar valores int
14       Bufer ubicacionCompartida = new BuferSincronizado();
15
16       System.out.printf( "40s%s\t\t%s\n%-40s%s\n\n", "Operacion",
```

Figura 23.23 | Dos subprocesos que manipulan un búfer sincronizado. (Parte 1 de 3).

```
17            "Bufer", "Ocupado", "---------", "------\t\t--------" );
18
19         // ejecuta las tareas Productor y Consumidor
20         aplicacion.execute( new Productor( ubicacionCompartida ) );
21         aplicacion.execute( new Consumidor( ubicacionCompartida ) );
22
23         aplicacion.shutdown();
24      } // fin de main
25   } // fin de la clase PruebaBuferCompartido2
```

```
Operacion                               Bufer          Ocupado
---------                               ------         --------

Productor escribe 1                     1              true

Productor trata de escribir.
Bufer lleno. Productor espera.          1              true


Consumidor lee 1                        1              false

Productor escribe 2                     2              true

Productor trata de escribir.
Bufer lleno. Productor espera.          2              true

Consumidor lee 2                        2              false

Productor escribe 3                     3              true

Consumidor lee 3                        3              false

Productor escribe 4                     4              true

Consumidor lee 4                        4              false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.         4              false

Productor escribe 5                     5              true

Consumidor lee 5                        5              false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.         5              false

Productor escribe 6                     6              true

Consumidor lee 6                        6              false

Productor escribe 7                     7              true

Consumidor lee 7                        7              false

Productor escribe 8                     8              true

Consumidor lee 8                        8              false

Productor escribe 9                     9              true
```

Figura 23.23 | Dos subprocesos que manipulan un búfer sincronizado. (Parte 2 de 3).

```
Consumidor lee 9                          9           false

Productor escribe 10                      10          true


Productor termino de producir
Terminando Productor
Consumidor lee 10                         10          false


Consumidor leyo valores, el total es 55
Terminando Consumidor
```

Figura 23.23 | Dos subprocesos que manipulan un búfer sincronizado. (Parte 3 de 3).

23.11 Subprocesamiento múltiple con GUIs

Las aplicaciones de Swing presentan un conjunto único de retos para la programación con subprocesamiento múltiple. Todas las aplicaciones de Swing tienen un solo subproceso, conocido como el **subproceso de despachamiento de eventos**, para manejar las interacciones con los componentes de la GUI de la aplicación. Las interacciones típicas incluyen actualizar los componentes de la GUI o procesar las acciones del usuario, como los clics del ratón. Todas las tareas que requieren interacción con la GUI de una aplicación se colocan en una cola de eventos y se ejecutan en forma secuencial, mediante el subproceso de despachamiento de eventos.

Los componentes de GUI de Swing no son seguros para los subprocesos; no se pueden manipular mediante varios subprocesos sin el riesgo de obtener resultados incorrectos. A diferencia de los demás ejemplos que se presentan en este capítulo, la seguridad de subprocesos en aplicaciones de GUI se logra, no mediante la sincronización de las acciones de los subprocesos, sino asegurando que se acceda a los componentes de Swing sólo desde un solo subproceso: el subproceso de despachamiento de eventos. A esta técnica se le conoce como **confinamiento de subprocesos**. Al permitir que sólo un subproceso acceda a los objetos que no son seguros para los subprocesos, se elimina la posibilidad de corrupción debido a que varios subprocesos accedan a estos objetos en forma concurrente.

Por lo general, basta con realizar cálculos simples en el subproceso de despachamiento de eventos, en secuencia con las manipulaciones de los componentes de GUI. Si una aplicación debe realizar un cálculo extenso en respuesta a una interacción con la interfaz del usuario, el subproceso de despachamiento de eventos no puede atender otras tareas en la cola de eventos, mientras se encuentre atado en ese cálculo. Esto hace que los componentes de la GUI pierdan su capacidad de respuesta. Es preferible manejar un cálculo extenso en un subproceso separado, con lo cual el subproceso de despachamiento de eventos queda libre para continuar administrando las demás interacciones con la GUI. Desde luego que, para actualizar la GUI con base en los resultados del cálculo, debemos actualizar la GUI desde el subproceso de despachamiento de eventos, en vez de hacerlo desde el subproceso trabajador que realizó el cálculo.

La clase `SwingWorker`

Java SE 6 cuenta con la clase `SwingWorker` (en el paquete `javax.swing`) para realizar cálculos extensos en un subproceso trabajador, y para actualizar los componentes de Swing desde el subproceso de despachamiento de eventos, con base en los resultados del cálculo. `SwingWorker` implementa a la interfaz `Runnable`, lo cual significa que un objeto `SwingWorker` se puede programar para ejecutarse en un subproceso separado. La clase `SwingWorker` proporciona varios métodos para simplificar la realización de cálculos en un subproceso trabajador, y hacer que estos resultados estén disponibles para mostrarlos en una GUI. En la figura 23.24 se describen algunos métodos comunes de `SwingWorker`.

23.11.1 Realización de cálculos en un subproceso trabajador

En el siguiente ejemplo, una GUI proporciona componentes para que el usuario escriba un número n y obtenga el n-ésimo número de Fibonacci, el cual calculamos mediante el uso del algoritmo recursivo que vimos en la sección 15.4. Como el algoritmo recursivo consume mucho tiempo para valores extensos, utilizamos un objeto

Método	Descripción
`doInBackground`	Define un cálculo extenso y se llama desde un subproceso trabajador.
`done`	Se ejecuta en el subproceso de despachamiento de eventos, cuando `doInBackground` regresa.
`execute`	Programa el objeto `SwingWorker` para que se ejecute en un subproceso trabajador.
`get`	Espera a que se complete el cálculo, y después devuelve el resultado del mismo (es decir, el valor de retorno de `doInBackground`).
`publish`	Envía resultados inmediatos del método `doInBackground` al método `process`, para procesarlos en el subproceso de despachamiento de eventos.
`process`	Recibe los resultados intermedios del método `publish` y los procesa en el subproceso de despachamiento de eventos.
`setProgress`	Establece la propiedad de progreso para notificar a cualquier componente de escucha de cambio de propiedades que esté en el subproceso de despachamiento de eventos, acerca de las actualizaciones en la barra de progreso.

Figura 23.24 | Métodos de uso común de `SwingWorker`.

`SwingWorker` para realizar el cálculo en un subproceso trabajador. La GUI también proporciona un conjunto separado de componentes que obtienen el siguiente número de Fibonacci en secuencia con cada clic de un botón, empezando con `fibonacci(1)`. Este conjunto de componentes realiza su cálculo corto directamente en el subproceso de despachamiento de eventos.

La clase `CalculadoraSegundoPlano` (figura 23.25) realiza el cálculo recursivo de Fibonacci en un subproceso trabajador. Esta clase extiende a `SwingWorker` (línea 8), sobrescribiendo a los métodos `doInBackground` y `done`. El método `doInBackground` (líneas 21 a 25) calcula el *n*-ésimo número de Fibonacci en un subproceso trabajador y devuelve el resultado. El método `done` (líneas 28 a 44) muestra el resultado en un objeto `JLabel`.

```
1  // Fig. 23.25: CalculadoraSegundoPlano.java
2  // Subclase de SwingWorker para calcular números de Fibonacci
3  // en un subproceso en segundo plano.
4  import javax.swing.SwingWorker;
5  import javax.swing.JLabel;
6  import java.util.concurrent.ExecutionException;
7
8  public class CalculadoraSegundoPlano extends SwingWorker< String, Object >
9  {
10    private final int n; // número de Fibonacci a calcular
11    private final JLabel resultadoJLabel; // JLabel para mostrar el resultado
12
13    // constructor
14    public CalculadoraSegundoPlano( int numero, JLabel etiqueta )
15    {
16       n = numero;
17       resultadoJLabel = etiqueta;
18    } // fin del constructor de CalculadoraSegundoPlano
19
20    // código que tarda mucho en ejecutarse, para ejecutarlo en un subproceso trabajador
21    public String doInBackground()
22    {
23       long nesimoFib = fibonacci( n );
```

Figura 23.25 | Subclase de `SwingWorker` para calcular números de Fibonacci en un subproceso en segundo plano. (Parte 1 de 2).

```
24          return String.valueOf( nesimoFib );
25       } // fin del método doInBackground
26
27       // código para ejecutar en el subproceso de despachamiento de eventos cuando regresa
         doInBackground
28       protected void done()
29       {
30          try
31          {
32             // obtiene el resultado de doInBackground y lo muestra
33             resultadoJLabel.setText( get() );
34          } // fin de try
35          catch ( InterruptedException ex )
36          {
37             resultadoJLabel.setText( "Se interrumpio mientras esperaba los resultados." );
38          } // fin de catch
39          catch ( ExecutionException ex )
40          {
41             resultadoJLabel.setText(
42                "Se encontro un error al realizar el calculo." );
43          } // fin de catch
44       } // fin del método done
45
46       // método recursivo fibonacci; calcula el n-ésimo número de Fibonacci
47       public long fibonacci( long numero )
48       {
49          if ( numero == 0 || numero == 1 )
50             return numero;
51          else
52             return fibonacci( numero - 1 ) + fibonacci( numero - 2 );
53       } // fin del método fibonacci
54    } // fin de la clase CalculadoraSegundoPlano
```

Figura 23.25 | Subclase de `SwingWorker` para calcular números de Fibonacci en un subproceso en segundo plano. (Parte 2 de 2).

Observe que `SwingWorker` es una clase genérica. En la línea 8, el primer parámetro de tipo es `String` y el segundo es `Object`. El primer parámetro de tipo indica el tipo devuelto por el método `doInBackground`; el segundo indica el tipo que se pasa entre los métodos `publish` y `process` para manejar los resultados intermedios. Como no utilizamos a `publish` o a `process` en este ejemplo, simplemente usamos `Object` como el segundo parámetro de tipo. En la sección 23.11.2 hablaremos sobre `publish` y `process`.

Un objeto `CalculadoraSegundoPlano` puede instanciarse a partir de una clase que controle una GUI. Un objeto `CalculadoraSegundoPlano` mantiene variables de instancia para un entero que representa el número de Fibonacci a calcular, y un objeto `Jlabel` que muestra los resultados del cálculo (líneas 10 y 11). El constructor de `CalculadoraSegundoPlano` (líneas 14 a 18) inicializa estas variables de instancia con los argumentos que se pasan al constructor.

Observación de ingeniería de software 23.3

Cualquier componente de la GUI que se manipule mediante métodos de `SwingWorker`, como los componentes que se actualizan a partir de los métodos `process` o `done`, se debe pasar al constructor de la subclase de `SwingWorker` para almacenarlo en el objeto de la subclase. Esto proporciona a estos métodos acceso a los componentes de la GUI que van a manipular.

Cuando se hace una llamada al método `execute` en un objeto `CalculadoraSegundoPlano`, el objeto se programa para ejecutarlo en un subproceso trabajador. El método `doInBackground` se llama desde el subproceso trabajador e invoca al método `fibonacci` (líneas 47 a 53), en donde recibe la variable de instancia `n` como argumento (línea 23). El método `fibonacci` utiliza la recursividad para calcular el número de Fibonacci de `n`. Cuando `fibonacci` regresa, el método `doInBackground` devuelve el resultado.

Una vez que `doInBackground` regresa, el método `done` se llama desde el subproceso de despachamiento de eventos. Este método trata de asignar al objeto `JLabel` que contiene el resultado del valor de retorno de `doInBackground`, mediante una llamada al método `get` para obtener este valor de retorno (línea 33). El método `get` espera a que el resultado este listo, en caso de ser necesario, pero como lo llamamos desde el método `done`, el cálculo se completará antes de que se llame a `get`. En las líneas 35 a 38 se atrapa una excepción `InterruptedException` si el subproceso actual se interrumpe mientras espera a que `get` regrese. En las líneas 39 a 43 se atrapa una excepción `ExecutionException`, que se lanza si ocurre una excepción durante el cálculo.

La clase `NumerosFibonacci` (figura 23.26) muestra una ventana que contiene dos conjuntos de componentes de GUI: uno para calcular un número de Fibonacci en un subproceso trabajador, y otro para obtener el siguiente número de Fibonacci en respuesta a la acción del usuario de oprimir un botón `JButton`. El constructor (líneas 38 a 109) coloca estos componentes en objetos `JPanel` con títulos separados. En las líneas 46 a 47 y 78 a 79 se agregan dos objetos `JLabel` un objeto `JTextField` y un objeto `JButton` al objeto `trabajadorJPanel` para que el usuario pueda introducir un entero, cuyo número de Fibonacci se calculará mediante el objeto `BackgroundWorker`. En las líneas 84 a 85 y 103 se agregan dos objetos `JLabel` y un objeto `JButton` al panel del subproceso de despachamiento de eventos, para permitir al usuario obtener el siguiente número de Fibonacci en la secuencia. Las variables de instancia `n1` y `n2` contienen los dos números de Fibonacci anteriores en la secuencia, y se inicializan con `0` y `1` respectivamente (líneas 29 y 30). La variable de instancia `cuenta` almacena el número en la secuencia que se calculó más recientemente, y se inicializa con `1` (línea 31). Al principio, los dos objetos `JLabel` muestran a cuenta y a `n2`, de manera que el usuario pueda ver el texto `Fibonacci de 1: 1` en el objeto `subprocesoEventosJPanel` cuando la GUI inicia.

```
1  // Fig. 23.26: NumerosFibonacci.java
2  // Uso de SwingWorker para realizar un cálculo extenso, en donde
3  // los resultados intermedios se muestran en una GUI.
4  import java.awt.GridLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JPanel;
10 import javax.swing.JLabel;
11 import javax.swing.JTextField;
12 import javax.swing.border.TitledBorder;
13 Import javax.swing.border.LineBorder;
14 import java.awt.Color;
15 import java.util.concurrent.ExecutionException;
16
17 public class NumerosFibonacci extends JFrame
18 {
19    // componentes para calcular el valor de Fibonacci de un número introducido por el
      usuario
20    private final JPanel trabajadorJPanel =
21       new JPanel( new GridLayout( 2, 2, 5, 5 ) );
22    private final JTextField numeroJTextField = new JTextField();
23    private final JButton iniciarJButton = new JButton( "Iniciar" );
24    private final JLabel fibonacciJLabel = new JLabel();
25
26    // componentes y variables para obtener el siguiente número de Fibonacci
27    private final JPanel subprocesoEventosJPanel =
28       new JPanel( new GridLayout( 2, 2, 5, 5 ) );
29    private int n1 = 0; // se inicializa con el primer número de Fibonacci
30    private int n2 = 1; // se inicializa con el segundo número de Fibonacci
31    private int cuenta = 1;
32    private final JLabel nJLabel = new JLabel( "Fibonacci de 1: " );
```

Figura 23.26 | Uso de `SwingWorker` para realizar un cálculo extenso, en donde los resultados intermedios se muestran en una GUI. (Parte 1 de 3).

```
33      private final JLabel nFibonacciJLabel =
34         new JLabel( String.valueOf( n2 ) );
35      private final JButton siguienteNumeroJButton = new JButton( "Siguiente numero");
36
37      // constructor
38      public NumerosFibonacci()
39      {
40         super( "Numeros de Fibonacci" );
41         setLayout( new GridLayout( 2, 1, 10, 10 ) );
42
43         // agrega componentes de GUI al panel de SwingWorker
44         trabajadorJPanel.setBorder( new TitledBorder(
45            new LineBorder( Color.BLACK ), "Con SwingWorker" ) );
46         trabajadorJPanel.add( new JLabel( "Obtener Fibonacci de:" ) );
47         trabajadorJPanel.add( numeroJTextField );
48         iniciarJButton.addActionListener(
49            new ActionListener()
50            {
51               public void actionPerformed( ActionEvent evento )
52               {
53                  int n;
54
55                  try
56                  {
57                     // obtiene la entrada del usuario como un entero
58                     n = Integer.parseInt( numeroJTextField.getText() );
59                  } // fin de try
60                  catch( NumberFormatException ex )
61                  {
62                     // muestra un mensaje de error si el usuario no
63                     // introdujo un entero
64                     fibonacciJLabel.setText( "Escriba un entero." );
65                     return;
66                  } // fin de catch
67
68                  // indica que ha empezado el cálculo
69                  fibonacciJLabel.setText( "Calculando..." );
70
71                  // crea una tarea para realizar el cálculo en segundo plano
72                  CalculadoraSegundoPlano tarea =
73                     new CalculadoraSegundoPlano( n, fibonacciJLabel );
74                  tarea.execute(); // ejecuta la tarea
75               } // fin del método actionPerformed
76            } // fin de la clase interna anónima
77         ); // fin de la llamada a addActionListener
78         trabajadorJPanel.add( iniciarJButton );
79         trabajadorJPanel.add( fibonacciJLabel );
80
81         // agrega componentes de GUI al panel del subproceso de despachamiento de eventos
82         subprocesoEventosJPanel.setBorder( new TitledBorder(
83            new LineBorder( Color.BLACK ), "Sin SwingWorker" ) );
84         subprocesoEventosJPanel.add( nJLabel );
85         subprocesoEventosJPanel.add( nFibonacciJLabel );
86         siguienteNumeroJButton.addActionListener(
87            new ActionListener()
88            {
89               public void actionPerformed( ActionEvent evento )
90               {
```

Figura 23.26 | Uso de `SwingWorker` para realizar un cálculo extenso, en donde los resultados intermedios se muestran en una GUI. (Parte 2 de 3).

```
91                  // calcula el número de Fibonacci después de n2
92                  int temp = n1 + n2;
93                  n1 = n2;
94                  n2 = temp;
95                  ++cuenta;
96
97                  // muestra el siguiente número de Fibonacci
98                  nJLabel.setText( "Fibonacci de " + cuenta + ": " );
99                  nFibonacciJLabel.setText( String.valueOf( n2 ) );
100              } // fin del método actionPerformed
101           } // fin de la clase interna anónima
102        ); // fin de la llamada a addActionListener
103        subprocesoEventosJPanel.add( siguienteNumeroJButton );
104
105        add( trabajadorJPanel );
106        add( subprocesoEventosJPanel );
107        setSize( 275, 200 );
108        setVisible( true );
109     } // fin del constructor
110
111     // el método main empieza la ejecución del programa
112     public static void main( String[] args )
113     {
114        NumerosFibonacci aplicacion = new NumerosFibonacci();
115        aplicacion.setDefaultCloseOperation( EXIT_ON_CLOSE );
116     } // fin de main
117  } // fin de la clase NumerosFibonacci
```

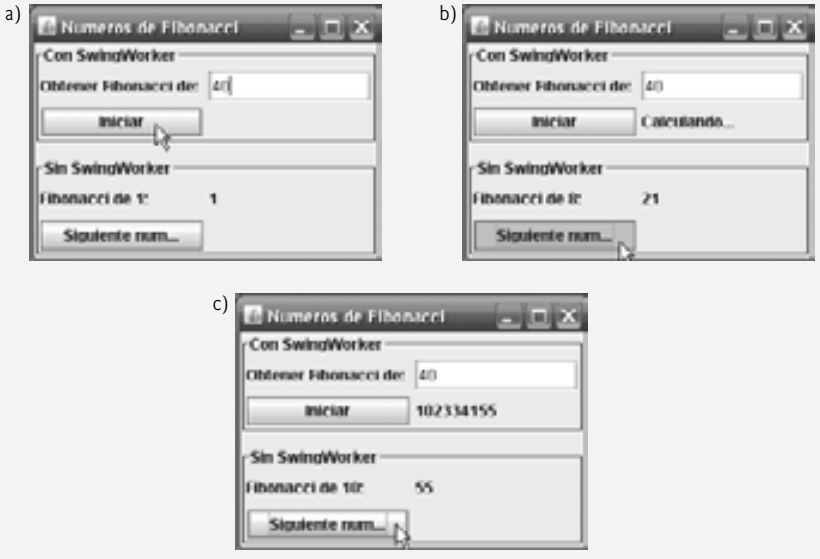


Figura 23.26 | Uso de `SwingWorker` para realizar un cálculo extenso, en donde los resultados intermedios se muestran en una GUI. (Parte 3 de 3).

En las líneas 48 a 77 se registra el manejador de eventos para el botón `iniciarJButton`. Si el usuario hace clic en este objeto `JButton`, en la línea 58 se obtiene el valor introducido en el objeto `numeroJTextField` y el programa intenta convertirlo en entero. En las líneas 72 y 73 se crea un nuevo objeto `CalculadoraSegundoPlano`, el cual recibe el valor introducido por el usuario y el objeto `fibonacciJLabel` que se utiliza para mostrar los resultados del cálculo. En la línea 74 se hace una llamada al método `execute` en el objeto `CalculadoraSegun-`

doPlano, y se programa para ejecutarlo en un subproceso trabajador separado. El método execute no espera a que el objeto CalculadoraSegundoPlano termine de ejecutarse. Regresa de inmediato, lo cual permite a la GUI continuar procesando otros eventos, mientras se realiza el cálculo.

Si el usuario hace clic en el objeto siguienteNumeroJButton que está en el objeto subprocesoEventosJPanel, se ejecuta el manejador de eventos registrado en las líneas 86 a 102. Los dos números de Fibonacci anteriores que están almacenados en n1 y n2 se suman y cuenta se incrementa para determinar el siguiente número en la secuencia (líneas 92 a 95). Después, en las líneas 98 y 99 se actualiza la GUI para mostrar el siguiente número. El código para realizar estos cálculos está escrito directamente en el método actionPerformed, por lo que estos cálculos se llevan a cabo en el subproceso de despachamiento de eventos. Al manejar estos cálculos cortos en el subproceso de despachamiento de eventos, la GUI no pierde capacidad de respuesta, como el algoritmo recursivo para calcular el valor de Fibonacci para un número extenso. Debido a que el cálculo del número de Fibonacci más extenso se realiza en un subproceso trabajador separado mediante el uso del objeto SwingWorker, es posible obtener el siguiente número de Fibonacci mientras el cálculo recursivo aún se está realizando.

23.11.2 Procesamiento de resultados inmediatos con SwingWorker

Hemos presentado un ejemplo en el que se utiliza la clase SwingWorker para ejecutar un proceso extenso en un subproceso en segundo plano, en donde la GUI se actualiza al terminar el proceso. Ahora presentaremos un ejemplo acerca de cómo actualizar la GUI con resultados intermedios antes de que el proceso extenso termine. En la figura 23.27 se presenta la clase CalculadoraPrimos, la cual extiende a SwingWorker para calcular los primeros *n* números primos en un subproceso trabajador. Además de los métodos doInBackground y done que se utilizaron en el ejemplo anterior, esta clase utiliza los métodos publish, process y setProgress de SwingWorker. En este ejemplo, el método publish envía números primos al método process a medida que se van encontrando, el método process muestra estos números primos en un componente de la GUI, y el método setProgress actualiza la propiedad de progreso. Más adelante le mostraremos cómo utilizar esta propiedad para actualizar un objeto JProgressBar.

```
1  // Fig. 23.27: CalculadoraPrimos.java
2  // Calcula los primeros n números primos, y muestra a medida que los va encontrando.
3  import javax.swing.JTextArea;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import javax.swing.SwingWorker;
7  import java.util.Random;
8  import java.util.List;
9  import java.util.concurrent.ExecutionException;
10
11 public class CalculadoraPrimos extends SwingWorker< Integer, Integer >
12 {
13    private final Random generador = new Random();
14    private final JTextArea intermedioJTextArea; // muestra los números primos
      encontrados
15    private final JButton obtenerPrimosJButton;
16    private final JButton cancelarJButton;
17    private final JLabel estadoJLabel; // muestra el estado del cálculo
18    private final boolean primos[]; // arreglo booleano para buscar números primos
19    private boolean detenido = false; // bandera que indica la cancelación
20
21    // constructor
22    public CalculadoraPrimos( int max, JTextArea intermedio, JLabel estado,
23       JButton obtenerPrimos, JButton cancel )
24    {
25       intermedioJTextArea = intermedio;
```

Figura 23.27 | Calcula los primeros *n* números primos, y los muestra a medida que los va encontrando. (Parte 1 de 3).

```
26          estadoJLabel = estado;
27          obtenerPrimosJButton = obtenerPrimos;
28          cancelarJButton = cancel;
29          primos = new boolean[ max ];
30
31          // inicializa todos los valores del arreglo primos con true
32          for ( int i = 0; i < max; i ++ )
33             primos[ i ] = true;
34       } // fin del constructor
35
36       // busca todos los números primos hasta max, usando la Criba de Eratóstenes
37       public Integer doInBackground()
38       {
39          int cuenta = 0; // la cantidad de números primos encontrados
40
41          // empezando en el tercer valor, itera a través del arreglo y pone
42          // false como el valor de cualquier número mayor que sea múltiplo
43          for ( int i = 2; i < primos.length; i++ )
44          {
45             if ( detenido ) // si se canceló un cálculo
46                return cuenta;
47             else
48             {
49                setProgress( 100 * ( i + 1 ) / primos.length );
50
51                try
52                {
53                   Thread.currentThread().sleep( generador.nextInt( 5 ) );
54                } // fin de try
55                catch ( InterruptedException ex )
56                {
57                   estadoJLabel.setText( "Se interrumpio subproceso Trabajador" );
58                   return cuenta;
59                } // fin de catch
60
61                if ( primos[ i ] ) // i es primo
62                {
63                   publish( i ); // hace a i disponible para mostrarlo en la lista de primos
64                   ++cuenta;
65
66                   for ( int j = i + i; j < primos.length; j += i )
67                      primos[ j ] = false; // i no es primo
68                } // fin de if
69             } // fin de else
70          } // fin de for
71
72          return cuenta;
73       } // fin del método doInBackground
74
75       // muestra los valores publicados en la lista de números primos
76       protected void process( List< Integer > valsPublicados )
77       {
78          for ( int i = 0; i < valsPublicados.size(); i++ )
79             intermedioJTextArea.append( valsPublicados.get( i ) + "\n" );
80       } // fin del método process
81
82       // código a ejecutar cuando se completa doInBackground
83       protected void done()
```

Figura 23.27 | Calcula los primeros *n* números primos, y los muestra a medida que los va encontrando. (Parte 2 de 3).

```
84      {
85         obtenerPrimosJButton.setEnabled( true ); // habilita el botón Obtener primos
86         cancelarJButton.setEnabled( false ); // deshabilita el botón Cancelar
87
88         int numPrimos;
89
90         try
91         {
92            numPrimos = get(); // obtiene el valor de retorno de doInBackground
93         } // fin de try
94         catch ( InterruptedException ex )
95         {
96            estadoJLabel.setText( "Se interrumpio mientras se esperaban los resultados." );
97            return;
98         } // fin de catch
99         catch ( ExecutionException ex )
100        {
101           estadoJLabel.setText( "Error al realizar el calculo." );
102           return;
103        } // fin de catch
104
105        estadoJLabel.setText( "Se encontraron " + numPrimos + " primos." );
106     } // fin del método done
107
108     // establece la bandera para dejar de buscar números primos
109     public void detenerCalculo()
110     {
111        detenido = true;
112     } // fin del método detenerCalculo
113  } // fin de la clase CalculadoraPrimos
```

Figura 23.27 | Calcula los primeros *n* números primos, y los muestra a medida que los va encontrando. (Parte 3 de 3).

La clase `CalculadoraPrimos` extiende a `SwingWorker` (línea 11); el primer parámetro de tipo indica el tipo de valor de retorno del método `doInBackground` y el segundo indica el tipo de los resultados intermedios que se pasan entre los métodos `publish` y `process`. En este caso, ambos parámetros de tipo son objetos `Integer`. El constructor (líneas 22 a 34) recibe como argumentos un entero que indica el límite superior de los números primos a localizar, un objeto `JTextArea` que se utiliza para mostrar los números primos en la GUI, un objeto `JButton` para iniciar un cálculo y otro para cancelarlo, y un objeto `JLabel` para mostrar el estado del cálculo.

En las líneas 32 y 33 se inicializan con `true` los elementos del arreglo `boolean` llamado `primos`. `CalculadoraPrimos` utiliza este arreglo y el algoritmo de la **Criba de Eratóstenes** (descrito en el ejercicio 7.27) para buscar todos los números primos menores que `max`. La Criba de Eratóstenes recibe una lista de números naturales de cualquier longitud y, empezando con el primer número primo, filtra todos sus múltiplos. Después avanza al siguiente número primo, que será el siguiente número que no esté filtrado todavía y elimina a todos sus múltiplos. Continúa hasta llegar al final de la lista y cuando se han filtrado todos los números que no son primos. En términos del algoritmo, empezamos con el elemento `2` del arreglo `boolean` y establecemos en `false` las celdas correspondientes a todos los valores que sean múltiplos de `2`, para indicar que pueden dividirse entre 2 y por ende, no son primos. Después avanzamos al siguiente elemento del arreglo, verificamos si es `true` y, de ser así, establecemos en `false` todos sus múltiplos para indicar que pueden dividirse entre el índice actual. Cuando se ha recorrido todo el arreglo de esta forma, todos los índices que contienen `true` son primos, ya que no tienen divisores.

En el método `doInBackground` (líneas 37 a 73), la variable de control `i` para el ciclo (líneas 43 a 70) controla el índice actual para implementar la Criba de Eratóstenes. En la línea 45 se evalúa la bandera `boolean` llamada `detenido`, la cual indica si el usuario hizo clic en el botón **Cancelar**. Si detenido es `true`, el método devuelve la cantidad de números primos encontrados hasta ese momento (línea 46) sin terminar el cálculo.

Si no se cancela el cálculo, en la línea 49 se hace una llamada al método `setProgress` para actualizar la propiedad de progreso con el porcentaje del arreglo que se ha recorrido hasta ese momento. En la línea 53 se pone en

inactividad el subproceso actual en ejecución durante un máximo de 4 milisegundos. En breve hablaremos sobre el por qué de esto. En la línea 61 se evalúa si el elemento del arreglo `primos` en el índice actual es `true` (y por ende, primo). De ser así, en la línea 63 se pasa el índice al método `publish` de manera que pueda mostrarse como resultado intermedio en la GUI, y en la línea 64 se incrementa el número de primos encontrados. En las líneas 66 y 67 se establecen en `false` todos los múltiplos del índice actual, para indicar que no son primos. Cuando se ha recorrido todo el arreglo `boolean`, el número de primos encontrados se devuelve en la línea 72.

En las líneas 76 a 80 se declara el método `process`, el cual se ejecuta en el subproceso de despachamiento de eventos y recibe su argumento `valsPublicados` del método `publish`. El paso de valores entre `publish` en el subproceso trabajador y `process` en el subproceso de despachamiento de eventos es asíncrono; `process` no se invoca necesariamente para cada una de las llamadas a `publish`. Todos los objetos `Integer` publicados desde la última llamada a `publish` se reciben como un objeto `List`, a través del método `process`. En las líneas 78 y 79 se itera a través de esta lista y se muestran los valores publicados en un objeto `JTextArea`. Como el cálculo en el método `doInBackground` progresa rápidamente, publicando valores con frecuencia, las actualizaciones al objeto `JTextArea` se pueden apilar en el subproceso de despachamiento de eventos, lo que puede provocar que la GUI tenga una respuesta lenta. De hecho, al buscar un número suficientemente largo de primos, el subproceso de despachamiento de eventos puede llegar a recibir tantas peticiones en una rápida sucesión para actualizar el objeto `JTextArea`, que el subproceso se quedará sin memoria en su cola de eventos. Ésta es la razón por la cual pusimos al subproceso trabajador en inactividad durante unos cuantos milisegundos, entre cada llamada potencial a `publish`. Se reduce la velocidad del cálculo lo suficiente como para permitir que el subproceso despachador de eventos se mantenga a la par con las peticiones para actualizar el objeto `JTextArea` con nuevos números primos, lo cual permite a la GUI actualizarse de manera uniforme y así puede permanecer con una capacidad de respuesta relativamente inmediata.

En las líneas 83 a 106 se define el método `done`. Cuando el cálculo termina o se cancela, el método `done` habilita el botón **Obtener primos** y deshabilita el botón **Cancelar** (líneas 85 y 86). En la línea 92 se obtiene el valor de retorno (el número de primos encontrados) del método `doInBackground`. En las líneas 94 a 103 se atrapan las excepciones lanzadas por el método `get` y se muestra un mensaje de error apropiado en el objeto `estadoJLabel`. Si no ocurren excepciones, en la línea 105 se establece el objeto `estadoJLabel` para indicar el número de primos encontrados.

En las líneas 109 a 112 se define el método `public detenerCalculo`, el cual se invoca cuando el usuario hace clic en el botón **Cancelar**. Este método establece la bandera `detenido` en la línea 111, de forma que `doInBackground` pueda regresar sin terminar su cálculo la próxima vez que evalúe esta bandera. Aunque `SwingWorker` cuenta con un método `cancel`, este método simplemente llama al método `interrupt` de `Thread` en el subproceso trabajador. Al utilizar la bandera `boolean` en vez del método `cancel`, podemos detener el cálculo limpiamente, devolver un valor de `doInBackground` y asegurar que se haga una llamada al método `done`, aun si el cálculo no se ejecutó hasta completarse, sin el riesgo de lanzar una excepción `InterruptedException` asociada con la acción de interrumpir el subproceso trabajador.

La clase `BuscarPrimos` (figura 23.28) muestra un objeto `JTextField` que permite al usuario escribir un número, un objeto `JButton` para empezar a buscar todos los números primos menores que ese número, y un objeto `JTextArea` para mostrar los números primos. Un objeto `JButton` permite al usuario cancelar el cálculo, y un objeto `JProgressBar` indica el progreso del cálculo. El constructor de `BuscarPrimos` (líneas 32 a 125) inicializa estos componentes y los muestra en un objeto `JFrame`, usando el esquema `BorderLayout`.

En las líneas 42 a 94 se registra el manejador de eventos para el objeto `obtenerPrimosJButton`. Cuando el usuario hace clic en este objeto `JButton`, en las líneas 47 a 49 se restablece el objeto `JProgressBar` y se borran los objetos `mostrarPrimosJTextArea` y `estadoJLabel`. En las líneas 53 a 63 se analiza el valor en el objeto `JtextField` y se muestra un mensaje de error si el valor no es un entero. En las líneas 66 a 68 se construye un nuevo objeto `CalculadoraPrimos`, el cual recibe como argumentos el entero que escribió el usuario, el objeto `mostrarPrimosJTextArea` para mostrar los números primos, el objeto `estadoJLabel` y los dos objetos `JButton`.

En las líneas 71 a 85 se registra un objeto `PropertyChangeListener` para el nuevo objeto `CalculadoraPrimos`, mediante el uso de una clase interna anónima. `PropertyChangeListener` es una interfaz del paquete `java.beans` que define un solo método, `propertyChange`. Cada vez que se invoca el método `setProgress` en un objeto `CalculadoraPrimos`, este objeto genera un evento `PropertyChangeEvent` para indicar que la propiedad de progreso ha cambiado. El método `propertyChange` escucha estos eventos. En la línea 78 se evalúa si un evento `PropertyChangeEvent` dado indica un cambio en la propiedad de progreso. De ser así, en la línea 80 se obtiene el nuevo valor de la propiedad y en la línea 81 se actualiza el objeto `JProgressBar` con el nuevo valor

de la propiedad de progreso. El objeto `JButton` "**Obtener primos**" se deshabilita (línea 88), de manera que sólo se pueda ejecutar un cálculo que actualice la GUI a la vez, y el objeto `JButton` "**Cancelar**" se habilita (línea 89) para permitir que el usuario detenga el cálculo antes de completarse. En la línea 91 se ejecuta el objeto `CalculadoraPrimos` para empezar a buscar números primos. Si el usuario hace clic en el objeto `cancelarJButton`, el manejador de eventos registrado en las líneas 107 a 115 llama al método `detenerCalculo` de `CalculadoraPrimos` (línea 112) y el cálculo regresa antes de terminar.

```
1  // Fig 23.28: BuscarPrimos.java
2  // Uso de un objeto SwingWorker para mostrar números primos y actualizar un objeto
3  // JProgressBar mientras se calculan los números primos.
4  import javax.swing.JFrame;
5  import javax.swing.JTextField;
6  import javax.swing.JTextArea;
7  import javax.swing.JButton;
8  import javax.swing.JProgressBar;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.ScrollPaneConstants;
13 import java.awt.BorderLayout;
14 import java.awt.GridLayout;
15 import java.awt.event.ActionListener;
16 import java.awt.event.ActionEvent;
17 import java.util.concurrent.ExecutionException;
18 import java.beans.PropertyChangeListener;
19 import java.beans.PropertyChangeEvent;
20
21 public class BuscarPrimos extends JFrame
22 {
23    private final JTextField primoMayor = new JTextField();
24    private final JButton obtenerPrimosJButton = new JButton( "Obtener primos" );
25    private final JTextArea mostrarPrimosJTextArea = new JTextArea();
26    private final JButton cancelarJButton = new JButton( "Cancelar" );
27    private final JProgressBar progresoJProgressBar = new JProgressBar();
28    private final JLabel estadoJLabel = new JLabel();
29    private CalculadoraPrimos calculadora;
30
31    // constructor
32    public BuscarPrimos()
33    {
34       super( "Busqueda de primos con SwingWorker" );
35       setLayout( new BorderLayout() );
36
37       // inicializa el panel para obtener un número del usuario
38       JPanel norteJPanel = new JPanel();
39       norteJPanel.add( new JLabel( "Buscar primos menores que: " ) );
40       primoMayor.setColumns( 5 );
41       norteJPanel.add( primoMayor );
42       obtenerPrimosJButton.addActionListener(
43          new ActionListener()
44          {
45             public void actionPerformed( ActionEvent e )
46             {
47                progresoJProgressBar.setValue( 0 ); // restablece JProgressBar
48                mostrarPrimosJTextArea.setText( "" ); // borra JTextArea
49                estadoJLabel.setText( "" ); // borra JLabel
```

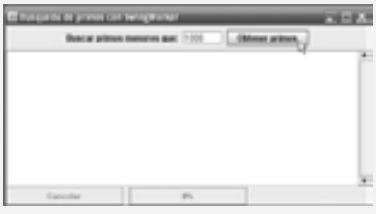
Figura 23.28 | Uso de un objeto `SwingWorker` para mostrar números primos y actualizar un objeto `JProgressBar` mientras se calculan los números primos. (Parte 1 de 3).

```
50
51                int numero;
52
53                try
54                {
55                    // obtiene la entrada del usuario
56                    numero = Integer.parseInt(
57                       primoMayor.getText() );
58                } // fin de try
59                catch ( NumberFormatException ex )
60                {
61                   estadoJLabel.setText( "Escriba un entero." );
62                   return;
63                } // fin de catch
64
65                // construye un nuevo objeto CalculadoraPrimos
66                calculadora = new CalculadoraPrimos( numero,
67                   mostrarPrimosJTextArea, estadoJLabel, obtenerPrimosJButton,
68                   cancelarJButton );
69
70                // escucha en espera de cambios en la propiedad de la barra de progreso
71                calculadora.addPropertyChangeListener(
72                   new PropertyChangeListener()
73                   {
74                      public void propertyChange( PropertyChangeEvent e )
75                      {
76                         // si la propiedad modificada es progreso (progress),
77                         // actualiza la barra de progreso
78                         if ( e.getPropertyName().equals( "progress" ) )
79                         {
80                            int nuevoValor = ( Integer ) e.getNewValue();
81                            progresoJProgressBar.setValue( nuevoValor );
82                         } // fin de if
83                      } // fin del método propertyChange
84                   } // fin de la clase interna anónima
85                ); // fin de la llamada a addPropertyChangeListener
86
87                // deshabilita el botón Obtener primos y habilita el botón Cancelar
88                obtenerPrimosJButton.setEnabled( false );
89                cancelarJButton.setEnabled( true );
90
91                calculadora.execute(); // ejecuta el objeto CalculadoraPrimos
92             } // fin del método ActionPerformed
93          } // fin de la clase interna anónima
94       ); // fin de la llamada a addActionListener
95       norteJPanel.add( obtenerPrimosJButton );
96
97       // agrega un objeto JList desplazable para mostrar el resultado del cálculo
98       mostrarPrimosJTextArea.setEditable( false );
99       add( new JScrollPane( mostrarPrimosJTextArea,
100         ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
101         ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER ) );
102
103      // inicializa un panel para mostrar a cancelarJButton,
104      // progresoJProgressBar y estadoJLabel
105      JPanel surJPanel = new JPanel( new GridLayout( 1, 3, 10, 10 ) );
106      cancelarJButton.setEnabled( false );
107      cancelarJButton.addActionListener(
```

Figura 23.28 | Uso de un objeto `SwingWorker` para mostrar números primos y actualizar un objeto `JProgressBar` mientras se calculan los números primos. (Parte 2 de 3).

```
108            new ActionListener()
109            {
110               public void actionPerformed( ActionEvent e )
111               {
112                  calculadora.detenerCalculo(); // cancela el cálculo
113               } // fin del método ActionPerformed
114            } // fin de la clase interna anónima
115         ); // fin de la llamada a addActionListener
116         surJPanel.add( cancelarJButton );
117         progresoJProgressBar.setStringPainted( true );
118         surJPanel.add( progresoJProgressBar );
119         surJPanel.add( estadoJLabel );
120
121         add( norteJPanel, BorderLayout.NORTH );
122         add( surJPanel, BorderLayout.SOUTH );
123         setSize( 350, 300 );
124         setVisible( true );
125      } // fin del constructor
126
127      // el método main empieza la ejecución del programa
128      public static void main( String[] args )
129      {
130         BuscarPrimos aplicacion = new BuscarPrimos();
131         aplicacion.setDefaultCloseOperation( EXIT_ON_CLOSE );
132      } // fin de main
133   } // fin de la clase BuscarPrimos
```

a)



b)

c)

Figura 23.28 | Uso de un objeto `SwingWorker` para mostrar números primos y actualizar un objeto `JProgressBar` mientras se calculan los números primos. (Parte 3 de 3).

23.12 Otras clases e interfaces en `java.util.concurrent`

La interfaz `Runnable` sólo proporciona la funcionalidad más básica para la programación con subprocesamiento múltiple. De hecho, esta interfaz tiene varias limitaciones. Suponga que un objeto `Runnable` encuentra un problema y trata de lanzar una excepción verificada. El método `run` no se declara para lanzar ninguna excepción, por lo que el problema se debe manejar dentro del objeto `Runnable`; la excepción no se puede pasar al subproceso que hizo la llamada. Ahora, suponga que un objeto `Runnable` va a realizar un cálculo extenso, y que la aplicación desea obtener el resultado de ese cálculo. El método `run` no puede devolver un valor, por lo que la aplicación debe utilizar datos compartidos para pasar el valor de vuelta al subproceso que hizo la llamada. Esto también implica la sobrecarga de sincronizar el acceso a los datos. Los desarrolladores de las APIs de concurrencia que se introdujeron en Java SE 5 reconocieron estas limitaciones, y crearon una nueva interfaz para corregirlas. La interfaz **`Callable`**

(del paquete `java.util.concurrent`) declara un solo método llamado `call`. Esta interfaz está diseñada para que sea similar a la interfaz `Runnable` (permitir que se realice una acción de manera concurrente en un subproceso separado), pero el método `call` permite al subproceso devolver un valor o lanzar una excepción verificada.

Es probable que una aplicación que crea un objeto `Callable` necesite ejecutar el objeto `Callable` de manera concurrente con otros objetos `Runnable` y `Callable`. La interfaz `ExecutorService` proporciona el método `submit`, el cual ejecuta un objeto `Callable` que recibe como argumento. El método `submit` devuelve un objeto de tipo `Future` (del paquete `java.util.concurrent`), la cual es una interfaz que representa al objeto `Callable` en ejecución. La interfaz `Future` declara el método `get` para devolver el resultado del objeto `Callable` y proporciona otros métodos para administrar la ejecución de un objeto `Callable`.

23.13 Conclusión

En este capítulo aprendió que, a través de la historia, la concurrencia se ha implementado con las primitivas de sistema operativo, disponibles sólo para los programadores experimentados de sistemas, pero que Java pone la concurrencia a nuestra disposición a través del lenguaje y las APIs. También aprendió que la JVM en sí crea subprocesos para ejecutar un programa, y que también puede crear subprocesos para realizar las tareas de mantenimiento, como la recolección de basura.

Hablamos sobre el ciclo de vida de un subproceso y los estados que éste puede ocupar durante su tiempo de vida. También hablamos sobre las prioridades de los subprocesos de Java, las cuales ayudan al sistema a programar los subprocesos para su ejecución. Aprendió que debe evitar manipular las prioridades de los subprocesos en Java directamente, y aprendió también acerca de los problemas asociados con las prioridades de los subprocesos, como el aplazamiento indefinido (conocido también como inanición).

Después presentamos la interfaz `Runnable`, que se utiliza para especificar que una tarea se puede ejecutar de manera concurrente con otras tareas. El método `run` de esta interfaz se invoca mediante el subproceso que ejecuta la tarea. Mostramos cómo ejecutar un objeto `Runnable`, asociándolo con un objeto de la clase `Thread`. Después mostramos cómo usar la interfaz `Executor` para administrar la ejecución de objetos `Runnable` a través de reservas de subprocesos, las cuales pueden reutilizar los subprocesos existentes para eliminar la sobrecarga de tener que crear un nuevo subproceso para cada tarea, y pueden mejorar el rendimiento al optimizar el número de procesadores, para asegurar que el procesador se mantenga ocupado.

Aprendió que cuando varios subprocesos comparten un objeto y uno o más de ellos modifica ese objeto, pueden ocurrir resultados indeterminados, a menos que el acceso al objeto compartido se administre de manera apropiada. Le mostramos cómo resolver este problema a través de la sincronización de subprocesos, la cual coordina el acceso a los datos compartidos por varios subprocesos concurrentes. Conoció varias técnicas para realizar la sincronización: primero con la clase integrada `ArrayBlockingQueue` (la cual se encarga de todos los detalles de sincronización por usted), después con los monitores integrados de Java y la palabra clave `synchronized`, y finalmente con las interfaces `Lock` y `Condition`.

Hablamos sobre el hecho de que las GUIs de Swing no son seguras para los subprocesos, por lo que todas las interacciones con (y las modificaciones a) la GUI deben realizarse en el subproceso despachador de eventos. También vimos los problemas asociados con la realización de cálculos extensos en el subproceso despachador de eventos. Después le mostramos cómo puede usar la clase `SwingWorker` de Java SE 6 para realizar cálculos extensos en subprocesos trabajadores. Aprendió a mostrar los resultados de un objeto `SwingWorker` en una GUI cuando se completa el cálculo, y a mostrar los resultados intermedios cuando el cálculo se está realizando.

Por último, hablamos sobre las interfaces `Callable` y `Future`, las cuales nos permiten ejecutar tareas que devuelvan resultados y a obtener esos resultados, respectivamente. En el capítulo 24, Redes, utilizaremos las técnicas de subprocesamiento múltiple que presentamos aquí para que nos ayuden a crear servidores con subprocesamiento múltiple, que puedan actuar con varios clientes de manera concurrente.

Resumen

Sección 23.1 Introducción

- A través de la historia, la concurrencia se ha implementado con primitivas de sistema operativo, disponibles sólo para los programadores de sistemas experimentados.

- El lenguaje de programación Ada, desarrollado por el Departamento de defensa de los Estados Unidos, hizo que las primitivas de concurrencia estuvieran disponibles ampliamente para los contratistas de defensa dedicados a la construcción de sistemas militares de comando y control.
- Java pone las primitivas de concurrencia a disposición del programador de aplicaciones, a través del lenguaje y de las APIs. El programador especifica que una aplicación contiene subprocesos de ejecución separados, en donde cada subproceso tiene su propia pila de llamadas a métodos y su propio contador, lo cual le permite ejecutarse concurrentemente con otros subprocesos, al mismo tiempo que comparte los recursos a nivel de aplicación (como la memoria) con estos otros subprocesos.
- Además de crear subprocesos para ejecutar un programa, la JVM también puede crear subprocesos para realizar tareas de mantenimiento, como la recolección de basura.

Sección 23.2 Estados de los subprocesos: ciclo de vida de un subproceso

- En cualquier momento dado, se dice que un subproceso se encuentra en uno de varios estados de subproceso.
- Un nuevo subproceso empieza su ciclo cuando en el estado *nuevo*. Permanece en este estado hasta que el programa inicia el subproceso, con lo cual se coloca en el estado *ejecutable*. Se considera que un subproceso en el estado *ejecutable* está ejecutando su tarea.
- Algunas veces, un subproceso *ejecutable* cambia al estado *en espera* mientras espera a que otro subproceso realice una tarea. Un subproceso *en espera* regresa al estado *ejecutable* sólo cuando otro subproceso notifica al subproceso en espera que puede continuar ejecutándose.
- Un subproceso *ejecutable* puede entrar al estado *en espera* sincronizado durante un intervalo específico de tiempo. Regresa al estado *ejecutable* cuando ese intervalo de tiempo expira, o cuando ocurre el evento que está esperando.
- Un subproceso *ejecutable* puede cambiar el estado en *espera sincronizado* si proporciona un intervalo de espera opcional cuando está esperando a que otro subproceso realice una tarea. Dicho subproceso regresará al estado *ejecutable* cuando otro subproceso se lo notifique o cuando expire el intervalo sincronizado.
- Un subproceso inactivo permanece en el estado *en espera* sincronizado durante un periodo designado de tiempo, después del cual regresa al estado *ejecutable*.
- Un subproceso *ejecutable* cambia al estado *bloqueado* cuando trata de realizar una tarea que no puede completarse inmediatamente, y debe esperar temporalmente hasta que se complete esa tarea. Al estado *bloqueado* cuando trata de realizar una tarea que no puede completarse inmediatamente, y debe esperar temporalmente hasta que se complete esa tarea. En ese punto, el subproceso bloqueado cambia al estado *ejecutable*, para poder continuar su ejecución. Un subproceso *bloqueado* no puede usar un procesador, aun si hay uno disponible.
- Un subproceso *ejecutable* entra al estado *terminado* cuando completa exitosamente su tarea, o termina de alguna otra forma (tal vez debido a un error).
- A nivel del sistema operativo, el estado *ejecutable* de Java generalmente abarca dos estados separados. Cuando un subproceso cambia por primera vez al estado *ejecutable* desde el estado *nuevo*, el subproceso se encuentra en el estado listo. Un subproceso *listo* entra al estado en *ejecución* cuando el sistema operativo lo asigna a un procesador; a esto también se le conoce como despachar el subproceso.
- En la mayoría de los sistemas operativos, a cada subproceso se le otorga una pequeña cantidad de tiempo del procesador (lo cual se conoce como quantum o intervalo de tiempo) en la que debe realizar su tarea. Cuando expira su *quantum*, el subproceso regresa al estado *listo* y el sistema operativo asigna otro subproceso al procesador.
- El proceso que utiliza un sistema operativo para determinar qué subproceso debe despachar se conoce como programación de subprocesos, y depende de las prioridades de los subprocesos.

Sección 23.3 Prioridades y programación de subprocesos

- Todo subproceso en Java tiene una prioridad de subproceso (de `MIN_PRIORITY` a `MAX_PRIORITY`), la cual ayuda al sistema operativo a determinar el orden en el que se programan los subprocesos.
- De manera predeterminada, cada subproceso recibe la prioridad `NORM_PRIORITY` (una constante de 5). Cada nuevo subproceso hereda la prioridad del subproceso que lo creó.
- La mayoría de los sistemas operativos permiten que los subprocesos con igual prioridad compartan un procesador con intervalo de tiempo.
- El trabajo del programador de subprocesos de un sistema operativo es determinar cuál subproceso se debe ejecutar a continuación.
- Cuando un subproceso de mayor prioridad entra al estado *listo*, el sistema operativo generalmente sustituye el subproceso actual en *ejecución* para dar preferencia al subproceso de mayor prioridad (una operación conocida como programación preferente).

- Dependiendo del sistema operativo, los subprocesos de mayor prioridad podrían posponer (tal vez de manera indefinida) la ejecución de los subprocesos de menor prioridad. Comúnmente, a dicho aplazamiento indefinido se le conoce, en forma más colorida, como inanición.

Sección 23.4 Creación y ejecución de subprocesos

- El medio preferido de crear aplicaciones en Java con subprocesamiento múltiple es mediante la implementación de la interfaz `Runnable` (del paquete `java.lang`). Un objeto `Runnable` representa una "tarea" que puede ejecutarse concurrentemente con otras tareas.
- La interfaz `Runnable` declara el método `run`, en el cual podemos colocar el código que define la tarea a realizar. El subproceso que ejecuta un objeto `Runnable` llama al método `run` para realizar la tarea.
- Un programa no terminará sino hasta que su último subproceso termine de ejecutarse; en este punto, la JVM también terminará.
- No podemos predecir el orden en el que se van a programar los subprocesos, aun si conocemos el orden en el que se crearon y se iniciaron.
- Aunque es posible crear subprocesos en forma explícita, se recomienda utilizar la interfaz `Executor` para administrar la ejecución de objetos `Runnable` de manera automática. Por lo general, un objeto `Executor` crea y administra un grupo de subprocesos, al cual se le denomina reserva de subprocesos, para ejecutar objetos `Runnable`.
- Los objetos `Executor` pueden reutilizar los subprocesos existentes para eliminar la sobrecarga de crear un nuevo subproceso para cada tarea, y pueden mejorar el rendimiento al optimizar el número de subprocesos, con lo cual se asegura que el procesador se mantenga ocupado.
- La interfaz `Executor` declara un solo método llamado `execute`, el cual acepta un objeto `Runnable` como argumento y lo asigna a uno de los subprocesos disponibles en la reserva. Si no hay subprocesos disponibles, el objeto `Executor` crea un nuevo subproceso, o espera a que haya uno disponible.
- La interfaz `ExecutorService` (del paquete `java.util.concurrent`) extiende a la interfaz `Executor` y declara varios métodos más para administrar el ciclo de vida de un objeto `Executor`.
- Un objeto que implementa a la interfaz `ExecutorService` se puede crear mediante el uso de los métodos `static` declarados en la clase `Executors` (del paquete `java.util.concurrent`).
- El método `newCachedThreadPool` de `Executors` devuelve un objeto `ExecutorService` que crea nuevos subprocesos, según los va necesitando la aplicación.
- El método `execute` de `ExecutorService` ejecuta el objeto `Runnable` que recibe como argumento en algún momento en el futuro. El método regresa inmediatamente después de cada invocación; el programa no espera a que termine cada tarea.
- El método `shutdown` de `ExecutorService` notifica al objeto `ExecutorService` para que deje de aceptar nuevas tareas, pero continúa ejecutando las tareas que ya se hayan enviado. Una vez que se han completado todos los objetos `Runnable` enviados anteriormente, el objeto `ExecutorService` termina.

Sección 23.5 Sincronización de subprocesos

- Cuando varios subprocesos comparten un objeto, y éste puede ser modificado por uno o más de los subprocesos, pueden ocurrir resultados indeterminados a menos que el acceso al objeto compartido se administre de manera apropiada. El problema puede resolverse si se da a un subproceso a la vez el *acceso exclusivo* al código que manipula al objeto compartido. Durante ese tiempo, otros subprocesos que deseen manipular el objeto deben mantenerse en espera. Cuando el subproceso con acceso exclusivo al objeto termina de manipularlo, a uno de los subprocesos que estaba en espera se le debe permitir que continúe ejecutándose. Este proceso, conocido como sincronización de subprocesos, coordina el acceso a los datos compartidos por varios subprocesos concurrentes.
- Al sincronizar los subprocesos, podemos asegurar que cada subproceso que accede a un objeto compartido excluye a los demás subprocesos de hacerlo en forma simultánea; a esto se le conoce como exclusión mutua.
- Una manera común de realizar la sincronización es mediante los monitores integrados en Java. Cada objeto tiene un monitor y un bloqueo de monitor. El monitor asegura que el bloqueo de monitor de su objeto se mantenga por un máximo de sólo un subproceso a la vez y, por ende, se puede utilizar para imponer la exclusión mutua.
- Si una operación requiere que el subproceso en ejecución mantenga un bloqueo mientras se realiza la operación, un subproceso debe adquirir el bloqueo para poder continuar con la operación. Otros subprocesos que traten de realizar una operación que requiera el mismo *bloqueo* permanecerán bloqueados hasta que el primer subproceso libere el bloqueo, punto en el cual los subprocesos *bloqueados* pueden tratar de adquirir el bloqueo.
- Para especificar que un subproceso debe mantener un bloqueo de monitor para ejecutar un bloque de código, el código debe colocarse en una instrucción `synchronized`. Se dice que dicho código está protegido por el bloqueo de monitor; un subproceso debe adquirir el bloqueo para ejecutar las instrucciones `synchronized`.

- Las instrucciones `synchronized` se declaran mediante la palabra clave `synchronized`:

```
synchronized ( objeto )
{
    instrucciones
} // fin de la instrucción synchronized
```

  en donde *objeto* es el objeto cuyo bloqueo de monitor se va a adquirir; generalmente, *objeto* es `this` si es el objeto en el que aparece la instrucción `synchronized`.
- Un método `synchronized` es equivalente a una instrucción `synchronized` que encierra el cuerpo completo de un método, y que utiliza a `this` como el objeto cuyo bloqueo de monitor se va a adquirir.
- La interfaz `ExecutorService` proporciona el método `awaitTermination` para obligar a un programa a esperar a que los subprocesos completen su ejecución. Este método devuelve el control al que lo llamó, ya sea cuando se completen todas las tareas que se ejecutan en el objeto `ExecutorService`, o cuando se agote el tiempo de inactividad especificado. Si todas las tareas se completan antes de que se agote el tiempo de `awaitTermination`, este método devuelve `true`; en caso contrario devuelve `false`. Los dos argumentos para `awaitTermination` representan un valor de límite de tiempo y una unidad de medida especificada con una constante de la clase `TimeUnit`.
- Podemos simular la atomicidad al asegurar que sólo un subproceso lleve a cabo un conjunto de operaciones al mismo tiempo. La atomicidad se puede lograr mediante el uso de la palabra clave `synchronized` para crear una instrucción o método `synchronized`.
- Al compartir datos inmutables entre subprocesos, debemos declarar los campos de datos correspondientes como `final`, para indicar que los valores de las variables no cambiarán una vez que se inicialicen.

Sección 23.6 Relación productor/consumidor sin sincronización

- En una relación productor/consumidor con subprocesamiento múltiple, un subproceso productor genera los datos y los coloca en un objeto compartido, llamado búfer. Un subproceso consumidor lee los datos del búfer.
- Las operaciones con los datos del búfer compartidos por un subproceso productor y un subproceso consumidor son dependientes del estado; las operaciones deben proceder sólo si el búfer se encuentra en el estado correcto. Si el búfer se encuentra en un estado en el que no esté completamente lleno, el productor puede producir; si el búfer se encuentra en un estado en el que no esté completamente vacío, el consumidor puede consumir.
- Los subprocesos con acceso a un búfer deben sincronizarse para asegurar que lo datos se escriban en el búfer, o se lean del búfer sólo si éste se encuentra en el estado apropiado. Si el productor que trata de colocar los siguientes datos en el búfer determina que éste se encuentra lleno, el subproceso productor debe esperar hasta que haya espacio. Si un subproceso consumidor encuentra el búfer vacío o que los datos anteriores ya se han leído, debe también esperar hasta que haya nuevos datos disponibles.

Sección 23.7 Relación productor/consumidor: `ArrayBlockingQueue`

- Java incluye una clase de búfer completamente implementada llamada `ArrayBlockingQueue` en el paquete `java.util.concurrent`, que implementa a la interfaz `BlockingQueue`. Esta interfaz extiende a la interfaz `Queue` y declara los métodos `put` y `take`, los equivalentes con bloqueo de los métodos `offer` y `poll` de `Queue`, respectivamente.
- El método `put` coloca un elemento al final del objeto `BlockingQueue`, y espera si la cola está llena. El método `take` elimina un elemento de la parte inicial del objeto `BlockingQueue`, y espera si la cola está vacía. Estos métodos hacen que la clase `ArrayBlockingQueue` sea una buena opción para implementar un búfer compartido. Debido a que el método `put` bloquea hasta que haya espacio en el búfer para escribir datos, y el método take bloquea hasta que haya nuevos datos para leer, el productor debe producir primero un valor, el consumidor sólo consume correctamente hasta después de que el productor escribe un valor, y el productor produce correctamente el siguiente valor (después del primero) sólo hasta que el consumidor lea el valor anterior (o primero).
- `ArrayBlockingQueue` almacena los datos compartidos en un arreglo. El tamaño de este arreglo se especifica como argumento para el constructor de `ArrayBlockingQueue`. Una vez creado, un objeto `ArrayBlockingQueue` tiene su tamaño fijo y no se expandirá para dar cabida a más elementos.

Sección 23.8 Relación productor/consumidor con sincronización

- Usted puede implementar su propio búfer compartido, usando la palabra clave `synchronized` y los métodos `wait`, `notify` y `notifyAll` de `Object`, que pueden usarse con condiciones para hacer que los subprocesos esperen cuando no puedan realizar sus tareas.
- Si un subproceso obtiene el bloqueo de monitor en un objeto, y después determina que no puede continuar con su tarea en ese objeto sino hasta que se cumpla cierta condición, el subproceso puede llamar al método `wait` de

`Object`; esto libera el bloqueo de monitor en el objeto, y el subproceso queda en el estado *en espera* mientras el otro subproceso trata de entrar a la(s) instrucción(es) o método(s) `synchronized` del objeto.
- Cuando un subproceso que ejecuta una instrucción (o método) `synchronized` completa o cumple con la condición en la que otro subproceso puede estar esperando, puede llamar al método `notify` de `Object` para permitir que un subproceso en espera cambie al estado *ejecutable* de nuevo. En este punto, el subproceso que cambió del estado en *espera* al estado *ejecutable* puede tratar de readquirir el bloqueo de monitor en el objeto.
- Si un subproceso llama a `notifyAll`, entonces todos los subprocesos que esperan el bloqueo de monitor se convierten en candidatos para readquirir el bloqueo (es decir, todos cambian al estado ejecutable).

Sección 23.9 Relación productor/consumidor: búferes delimitados
- No podemos hacer suposiciones acerca de las velocidades relativas de los subprocesos concurrentes; las interacciones que ocurren con el sistema operativo, la red, el usuario y otros componentes pueden hacer que los subprocesos operen a distintas velocidades. Cuando esto ocurre, los subprocesos esperan.
- Para minimizar la cantidad de tiempo de espera para los subprocesos que comparten recursos y operan a las mismas velocidades promedio, podemos implementar un bufer delimitado. Si el productor produce temporalmente valores con más rapidez de la que el consumidor pueda consumirlos, el productor puede escribir otros valores en el espacio adicional del búfer (si hay disponible). Si el consumidor consume con más rapidez de la que el productor produce nuevos valores, el consumidor puede leer valores adicionales (si los hay) del búfer.
- La clave para usar un búfer delimitado con un productor y un consumidor que operan aproximadamente a la misma velocidad es proporcionar al búfer suficientes ubicaciones para que pueda manejar la producción "extra" anticipada.
- La manera más simple de implementar un búfer delimitado es utilizar un objeto `ArrayBlockingQueue` para el búfer, de manera que se haga cargo de todos los detalles de la sincronización por nosotros.

Sección 23.10 Relación productor/consumidor: las interfaces `Lock` y `Condition`
- Las interfaces `Lock` y `Condition`, que se introdujeron en Java SE 5, proporcionan a los programadores un control más preciso sobre la sincronización de los subprocesos, pero son más complicadas de usar.
- Cualquier objeto puede contener una referencia a un objeto que implemente a la interfaz `Lock` (del paquete `java.util.concurrent.locks`). Un subproceso llama al método `lock` de `Lock` para adquirir el bloqueo. Una vez que un subproceso obtiene un objeto `Lock`, este objeto no permitirá que otro subproceso obtenga el `Lock` sino hasta que el primer subproceso lo libere (llamando al método `unlock` de `Lock`).
- Si varios subprocesos tratan de llamar al método `lock` en el mismo objeto `Lock` y al mismo tiempo, sólo uno de estos subprocesos puede obtener el bloqueo; todos los demás se colocan en el estado *en espera* de ese bloqueo. Cuando un subproceso llama al método `unlock`, se libera el bloqueo sobre el objeto y un subproceso en espera que intente bloquear el objeto puede continuar.
- La clase `ReentrantLock` (del paquete `java.util.concurrent.locks`) es una implementación básica de la interfaz `Lock`.
- El constructor de `ReentrantLock` recibe un argumento `boolean`, el cual especifica si el bloqueo tiene una política de equidad. Si el argumento es `true`, la política de equidad de `ReentrantLock` es: "el subproceso con más tiempo de espera adquirirá el bloqueo cuando esté disponible". Dicha política de equidad garantiza que nunca ocurra el aplazamiento indefinido (también conocido como inanición). Si el argumento de la política de equidad se establece en `false`, no hay garantía en cuanto a cuál subproceso en espera adquirirá el bloqueo cuando esté disponible.
- Si un subproceso que posee un objeto `Lock` determina que no puede continuar con su tarea hasta que se cumpla cierta condición, el subproceso puede esperar en base a un objeto de condición. El uso de objetos `Lock` nos permite declarar de manera explícita los objetos de condición sobre los cuales un subproceso tal vez tenga que esperar.
- Los objetos de condición se asocian con un objeto `Lock` específico y se crean mediante una llamada al método `newCondition` de `Lock`, el cual devuelve un objeto que implementa a la interfaz `Condition` (del paquete `java.util.concurrent.locks`). Para esperar en base a un objeto de condición, el subproceso puede llamar al método `await` de `Condition`. Esto libera de inmediato el objeto `Lock` asociado, y coloca al subproceso en el estado *en espera*, en base a ese objeto `Condition`. Así, otros subprocesos pueden tratar de obtener el objeto `Lock`.
- Cuando un subproceso *ejecutable* completa una tarea y determina que el subproceso en espera puede ahora continuar, el subproceso *ejecutable* puede llamar al método `signal` de `Condition` para permitir que un subproceso en el estado *en espera* de ese objeto `Condition` regrese al estado *ejecutable*. En este punto, el subproceso que cambió del estado en espera al estado *ejecutable* puede tratar de readquirir el objeto `Lock`.
- Si hay varios subprocesos en un estado *en espera* de un objeto `Condition` cuando se hace la llamada a `signal`, la implementación predeterminada de Condition indica al subproceso con más tiempo de espera que debe cambiar al estado *ejecutable*.

- Si un subproceso llama al método `signalAll` de `Condition`, entonces todos los subprocesos que esperan esa condición cambian al estado *ejecutable* y se convierten en candidatos para readquirir el objeto `Lock`.
- Cuando un subproceso termina con un objeto compartido, debe llamar al método `unlock` para liberar al objeto `Lock`.
- En algunas aplicaciones, el uso de objetos `Lock` y `Condition` puede ser preferible a utilizar la palabra clave `synchronized`. Los objetos `Lock` nos permiten interrumpir a los subprocesos en espera, o especificar un tiempo límite para esperar a adquirir un bloqueo, lo cual no es posible si se utiliza la palabra clave `synchronized`. Además, un objeto `Lock` no está restringido a ser adquirido y liberado en el mismo bloque de código, lo cual es el caso con la palabra clave `synchronized`.
- Los objetos `Condition` nos permiten especificar varios objetos de condición, en base a los cuales los subprocesos pueden esperar. Por ende, es posible indicar a los subprocesos en espera que un objeto de condición específico es ahora verdadero, llamando a `signal` o `signalAll` en ese objeto `Condition`. Con la palabra clave `synchronized`, no hay forma de indicar de manera explícita la condición en la cual esperan los subprocesos y, por lo tanto, no hay forma de notificar a los subprocesos en espera de una condición específica que pueden continuar, sin también indicarlo a los subprocesos que están en espera de otras condiciones.

Sección 23.11 Subprocesamiento múltiple con GUIs

- Las aplicaciones de Swing tienen un subproceso, conocido como el subproceso de despachamiento de eventos, para manejar las interacciones con los componentes de la GUI de la aplicación. Todas las tareas que requieren interacción con la GUI de una aplicación se colocan en una cola de eventos y se ejecutan en forma secuencial, mediante el subproceso de despachamiento de eventos.
- Los componentes de GUI de Swing no son seguros para los subprocesos. La seguridad de subprocesos en aplicaciones de GUI se logra asegurando que se acceda a los componentes de Swing sólo desde el subproceso de despachamiento de eventos. A esta técnica se le conoce como confinamiento de subprocesos.
- Si una aplicación debe realizar un cálculo extenso en respuesta a una interacción con la interfaz del usuario, el subproceso de despachamiento de eventos no puede atender otras tareas en la cola de eventos, mientras se encuentre atado en ese cálculo. Esto hace que los componentes de la GUI pierdan su capacidad de respuesta. Es preferible manejar un cálculo extenso en un subproceso separado, con lo cual el subproceso de despachamiento de eventos queda libre para continuar administrando las demás interacciones con la GUI.
- Java SE 6 cuenta con la clase `SwingWorker` (en el paquete `javax.swing`), que implementa a la interfaz `Runnable`, para realizar cálculos extensos en un subproceso trabajador, y para actualizar los componentes de Swing desde el subproceso de despachamiento de eventos, con base en los resultados del cálculo.
- Para utilizar las herramientas de `SwingWorker`, cree una clase que extienda a `SwingWorker` y sobrescriba los métodos `doInBackground` y `done`. El método `doInBackground` realiza el cálculo y devuelve el resultado. El método `done` muestra los resultados en la GUI.
- `SwingWorker` es una clase genérica. Su primer parámetro de tipo indica el tipo devuelto por el método `doInBackground`; el segundo indica el tipo que se pasa entre los métodos `publish` y `process` para manejar los resultados intermedios.
- El método `doInBackground` se llama desde un subproceso trabajador. Después de que `doInBackground` regresa, el método `done` se llama desde el subproceso de despachamiento de eventos para mostrar los resultados.
- Una excepción `ExecutionException` se lanza si ocurre una excepción durante el cálculo.
- `SwingWorker` también cuenta con los métodos `publish`, `process` y `setProgress`. El método `publish` envía en forma repetida los resultados inmediatos al método `process`, el cual muestra los resultados en un componente de la GUI. El método `setProgress` actualiza la propiedad de progreso.
- El método `progress` se ejecuta en el subproceso de despachamiento de eventos y recibe datos del método `publish`. El paso de valores entre `publish` en el subproceso trabajador y `process` en el subproceso de despachamiento de eventos es asíncrono; `process` no se invoca necesariamente para cada llamada a `publish`.
- `PropertyChangeListener` es una interfaz del paquete `java.beans` que define un solo método, `propertyChange`. Cada vez que se invoca el método `setProgress`, se genera un evento `PropertyChangedEvent` para indicar que la propiedad de progreso ha cambiado.

Sección 23.12 Otras clases e interfaces en `java.util.concurrent`

- La interfaz `Callable` (del paquete `java.util.concurrent`) declara un solo método llamado `call`. Esta interfaz está diseñada para que sea similar a la interfaz `Runnable` (permitir que se realice una acción de manera concurrente en un subproceso separado), pero el método `call` permite al subproceso devolver un valor o lanzar una excepción verificada.

- Es probable que una aplicación que crea un objeto `Callable` necesite ejecutar el objeto `Callable` de manera concurrente con otros objetos `Runnable` y `Callable`. La interfaz `ExecutorService` proporciona el método `submit`, el cual ejecuta un objeto `Callable` que recibe como argumento.
- El método `submit` devuelve un objeto de tipo `Future` (del paquete `java.util.concurrent`), que representa al objeto `Callable` en ejecución. La interfaz `Future` declara el método `get` para devolver el resultado del objeto `Callable` y proporciona otros métodos para administrar la ejecución de un objeto `Callable`.

Terminología

adquirir el bloqueo
aplazamiento indefinido
`ArrayBlockingQueue`, clase
`await`, método de la interfaz `Condition`
`awaitTermination`, método de la interfaz `ExecutorService`
`BlockingQueue`, interfaz
bloqueado, estado
bloqueo de monitor
bloqueo intrínseco
búfer
búfer circular
búfer delimitado
`call`, método de la interfaz `Callable`
`Callable`, interfaz
cola de prioridad multinivel
concurrencia
`Condition`, interfaz
confinamiento de subprocesos
consumidor
datos mutables
dependencia de estados
despachamiento de un subproceso
ejecutable, estado
en ejecución, estado
en espera sincronizado, estado
en espera, estado
estado de un subproceso
exclusión mutua
`execute`, método de la interfaz `Executor`
`Executor`, interfaz
`Executors`, clase
`ExecutorService`, interfaz
`Future`, interfaz
`get`, método de la interfaz `Future`
`IllegalMonitorStateException`, clase
inanición
interbloqueo
`interrupt`, método de la clase `Thread`
`InterruptedException`, clase
intervalo de inactividad
intervalo de tiempo
`java.util.concurrent`, paquete
`java.util.concurrent.locks`, paquete
listo, estado
`Lock`, interfaz
`lock`, método de la interfaz `Lock`
monitor
`newCachedThreadPool`, método de la clase `Executors`
`newCondition`, método de la interfaz `Lock`
`notify`, método de la clase `Object`
`notifyAll`, método de la clase `Object`
nuevo, estado
objeto de condición
operación atómica
operaciones en paralelo
política de equidad de un bloqueo
prioridad de subprocesos
productor
programación cíclica (round-robin)
programación concurrente
programación de subprocesos
programación preferente
programador de subprocesos
`propertyChange`, método de la interfaz `PropertyChangeListener`
`PropertyChangeListener`, interfaz
protegido por un bloqueo
`put`, método de la interfaz `BlockingQueue`
quantum
recolección de basura
`ReentrantLock`, clase
relación productor/consumidor
reserva de subprocesos
`run`, método de la interfaz `Runnable`
`Runnable`, interfaz
seguro para los subprocesos
`shutdown`, método de la clase `ExecutorService`
`signal`, método de la clase `Condition`
`signalAll`, método de la clase `Condition`
sincronización
sincronización de subprocesos
`size`, método de la clase `ArrayBlockingQueue`
`sleep`, método de la clase `Thread`
`submit`, método de la clase `ExecutorService`
subprocesamiento múltiple
subproceso
subproceso consumidor
subproceso de despachamiento de eventos
subproceso inactivo
subproceso principal
subproceso productor
`SwingWorker`, clase
`synchronized`, instrucción
`synchronized`, método
`synchronized`, palabra clave

`take`, método de la interfaz `BlockingQueue`
terminado, estado
`Thread`, clase
`unlock`, método de la interfaz `Lock`
valor pasado
`wait`, método de la clase `Object`

Ejercicios de autoevaluación

23.1 Complete las siguientes oraciones:

a) C y C++ son lenguajes con subprocesamiento _____, mientras que Java es un lenguaje con subprocesamiento _____.
b) Un subproceso entra al estado *terminado* cuando _____.
c) Para detenerse durante cierto número designado de milisegundos y reanudar su ejecución, un subproceso debe llamar al método _____ de la clase _____.
d) El método _____ de la clase `Condition` pasa a un solo subproceso en el estado *en espera* de un objeto, al estado *ejecutable*.
e) El método _____ de la clase `Condition` pasa a todos los subprocesos en el estado *en espera* de un objeto, al estado *ejecutable*.
f) Un subproceso _____ entra al estado _____ cuando completa su tarea, o cuando termina de alguna otra forma.
g) Un subproceso *ejecutable* puede entrar al estado _____ durante un intervalo específico.
h) A nivel del sistema operativo, el estado *ejecutable* en realidad abarca dos estados separados, _____ _____ y _____.
i) Los objetos `Runnable` se ejecutan usando una clase que implementa a la interfaz _____.
j) El método _____ de `ExecutorService` termina cada subproceso en un objeto `ExecutorService` tan pronto como termina de ejecutar su objeto `Runnable` actual, si lo hay.
k) Un subproceso puede llamar al método _____ en un objeto `Condition` para liberar el objeto `Lock` asociado, y colocar ese subproceso en el estado _____.
l) En una relación _____, la porción correspondiente al _____ de una aplicación genera datos y los almacena en un objeto compartido, y la porción correspondiente al _____ de una aplicación lee datos del objeto compartido.
m) La clase _____ implementa a la interfaz `BlockingQueue`, usando un arreglo.
n) La palabra clave _____ indica que sólo se debe ejecutar un subproceso a la vez en un objeto.

23.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un subproceso no es *ejecutable* si ha terminado.
b) Un subproceso *ejecutable* de mayor prioridad tiene preferencia sobre los subprocesos de menor prioridad.
c) Algunos sistemas operativos utilizan el intervalo de tiempo con subprocesos. Por lo tanto, pueden permitir que los subprocesos desplacen a otros subprocesos de la misma prioridad.
d) Cuando expira el quantum de un subproceso, éste regresa al estado *ejecutable*, a medida que el sistema operativo asigna el subproceso a un procesador.
e) En un sistema con un solo procesador sin intervalo de tiempo, cada subproceso en un conjunto de subprocesos de igual prioridad (sin otros subprocesos presentes) se ejecuta hasta terminar, antes de que otros subprocesos de igual prioridad tengan oportunidad de ejecutarse.

Respuestas a los ejercicios de autoevaluación

23.1 a) simple, múltiple. b) termina su método `run`. c) `sleep`, `Thread`. d) `signal`. e) `signalAll`, f) *ejecutable*, *terminado*. g) *en espera sincronizado*. h) *listo*, *en ejecución*. i) `Executor`. j) `shutdown`. k) `await`, *en espera*. l) productor/consumidor, productor, consumidor. m) `ArrayBlockingQueue`. n) `synchronized`.

23.2 a) Verdadero. b) Verdadero. c) Falso. El intervalo de tiempo permite a un subproceso ejecutarse hasta que expira su porción de tiempo (o quantum). Después pueden ejecutarse otros subprocesos de igual prioridad. d) Falso. Cuando expira el quantum de un subproceso, éste regresa al estado *listo* y el sistema operativo asigna otro subproceso al procesador. e) Verdadero.

Ejercicios

23.3 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) El método `sleep` no consume tiempo del procesador cuando un subproceso está inactivo.
b) Al declarar un método como `synchronized` se garantiza que no pueda ocurrir el interbloqueo.
c) Una vez que un subproceso obtiene un objeto `Lock`, éste no permitirá que otro subproceso obtenga el bloqueo sino hasta que el primer subproceso lo libere.
d) Los componentes de Swing son seguros para los subprocesos.

23.4 Defina cada uno de los siguientes términos.
a) subproceso
b) subprocesamiento múltiple
c) estado *ejecutable*
d) estado *en espera* sincronizado
e) programación preferente
f) interfaz `Runnable`
g) método `notifyAll`
h) relación productor/consumidor
i) quantum

23.5 Describa cada uno de los siguientes términos en el contexto de los mecanismos de subprocesamiento en Java:
a) `synchronized`
b) productor
c) consumidor
d) `wait`
e) `notify`
f) `Lock`
g) `Condition`

23.6 Enliste las razones para entrar al estado *bloqueado*. Para cada una de éstas, describa la forma en que el programa comúnmente sale del estado *bloqueado* y entra al estado *ejecutable*.

23.7 Dos problemas que pueden ocurrir en sistemas que permiten a los subprocesos esperar son: el interbloqueo, en el cual uno o más subprocesos esperarán para siempre un evento que no puede ocurrir, y el aplazamiento indefinido, en donde uno o más subprocesos se retrasarán durante cierto tiempo, sin saber cuánto. Dé un ejemplo de cómo cada uno de estos problemas puede ocurrir en los programas de Java con subprocesamiento múltiple.

23.8 Escriba un programa para rebotar una pelota azul dentro de un objeto `JPanel`. La pelota deberá empezar a moverse con un evento `mousePressed`. Cuando la pelota pegue en el borde del objeto `JPanel`, deberá rebotar y continuar en la dirección opuesta. La pelota debe actualizarse mediante el uso de un objeto `Runnable`.

23.9 Modifique el programa del ejercicio 23.8 para agregar una nueva pelota cada vez que el usuario haga clic con el ratón. Proporcione un mínimo de 20 pelotas. Seleccione al azar el color para cada nueva pelota.

23.10 Modifique el programa del ejercicio 23.9 para agregar sombras. A medida que se mueva una pelota, dibuje un óvalo relleno de color negro en la parte inferior del objeto `JPanel`. Tal vez sería conveniente agregar un efecto tridimensional, incrementando o decrementando el tamaño de cada pelota cuando ésta pegue en el borde del subprograma.