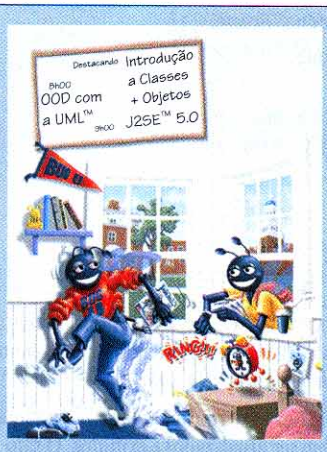
23

Multithreading



Não bloqueie o modo de pesquisa.
—Charles Sanders Peirce

Uma pessoa com um relógio sabe que horas são; uma pessoa com dois relógios nunca está segura.
—Provérbio

Aprenda a trabalhar e a esperar.
—Henry Wadsworth Longfellow

A mais geral definição da beleza ...Multiplicidade na Unidade.
—Samuel Taylor Coleridge

O mundo está mudando tão rápido hoje que um homem que diz que algo não pode ser feito em geral é interrompido por alguém fazendo esse algo.
—Elbert Hubbard

OBJETIVOS

Neste capítulo você aprenderá:

- O que são as threads e por que elas são úteis.
- Como as threads permitem gerenciar atividades concorrentes.
- O ciclo de vida de uma thread.
- Prioridades e agendamento de threads.
- Como criar e executar `Runnable`s.
- Sincronização de threads.
- O que são relacionamentos produtor/consumidor e como eles são implementados com multithreading.
- Como exibir a saída de múltiplas threads em uma GUI Swing.
- Sobre `Callable` e `Future`.

Sumário

23.1 Introdução

Seria interessante se pudéssemos fazer uma coisa por vez, e fazê-la bem, mas em geral isso é difícil. O corpo humano realiza uma grande variedade de operações **paralelamente** — ou, como diremos por todo este capítulo, **concorrentemente**. A respiração, a circulação sanguínea, a digestão, o pensamento e a locomoção, por exemplo, podem ocorrer simultaneamente. Todos os sentidos — visão, tato, olfato, paladar e audição — podem ocorrer ao mesmo tempo. Os computadores também realizam operações concorrentemente. É comum aos computadores pessoais compilar um programa, enviar um arquivo para uma impressora e receber mensagens de correio eletrônico em uma rede concorrentemente. Apenas os computadores que têm múltiplos processadores podem, de fato, executar operações concorrentemente. Em computadores de um único processador, os sistemas operacionais utilizam várias técnicas para simular a concorrência, mas em computadores como esses uma única operação pode executar por vez.

A maioria das linguagens de programação não permite que os programadores especifiquem atividades concorrentes. Em geral, elas fornecem apenas instruções de controle que permitem que os programadores realizem uma ação por vez, avançando para a próxima ação depois de a anterior ser concluída. Historicamente, a concorrência foi implementada com as primitivas de sistemas operacionais disponíveis apenas para programadores de sistemas experientes.

A linguagem de programação *Ada*, desenvolvida pelo Departamento de Defesa dos Estados Unidos tornou primitivos de concorrência amplamente disponíveis para as empresas contratadas do Departamento de Defesa que estavam construindo sistemas de controle e comando militar. Entretanto, a tecnologia Ada não foi amplamente utilizada em universidades e indústria comercial.

O Java disponibiliza a concorrência para o programador de aplicativos por meio de suas APIs. O programador especifica os aplicativos que contêm **threads de execução**, em que cada thread designa uma parte de um programa que pode executar concorrentemente com outras threads. Essa capacidade, chamada **multithreading**, fornece capacidades poderosas para o programador de Java não disponíveis no núcleo das linguagens C e C++ em que o Java é baseado.

Dica de desempenho 23.1

Um problema com aplicativos de uma única thread é que atividades longas devem ser concluídas antes que outras atividades iniciem. Em um aplicativo com múltiplas threads, as threads podem ser distribuídas por múltiplos processadores (se estes estiverem disponíveis), de modo que múltiplas tarefas são realizadas concorrentemente e o aplicativo pode operar de modo mais eficiente. O multithreading também pode aumentar o desempenho em sistemas de um único processador que simula a concorrência — quando uma thread não puder prosseguir, outra pode utilizar o processador.

Dica de portabilidade 23.1

Ao contrário das linguagens que não têm capacidades de multithreading integradas (como C e C++) e, portanto, devem fazer chamadas não-portáveis para primitivos de multithreading do sistema operacional, o Java inclui primitivas de multithreading como parte da própria linguagem e de suas bibliotecas. Isso facilita a manipulação de threads de maneira portável entre plataformas.

Discutiremos muitas aplicações da **programação de processos concorrentes**. Por exemplo, quando os programas fazem download de arquivos grandes, como clipes de áudio ou videoclipes, pela Internet, os usuários podem não querer esperar a finalização do download de um longo clipe antes de iniciar a reprodução. Para resolver esse problema, podemos colocar múltiplas threads para trabalhar — uma thread faz o download do clipe e a outra o reproduz. Essas atividades prosseguem concorrentemente. Para evitar a reprodução instável, vamos **sincronizar** as threads de modo que a thread do player não inicie até que haja uma quantidade suficiente do clipe na memória para manter a thread do player ocupado.

Outro exemplo de multithreading é a **coleta de lixo** do Java. As linguagens de programação C e C++ exigem que o programador reivindique memória dinamicamente alocada de modo explícito. O Java fornece uma **thread coletora de lixo** que reivindica a memória que não é mais necessária.

Escrever programas multithreaded pode ser difícil. Embora a mente humana possa realizar funções simultaneamente, as pessoas acham difícil alternar linhas de pensamento paralelas. Para ver por que multithreading pode ser difícil de programar e entender, faça a seguinte experiência: abra três livros na página 1 e tente lê-los concorrentemente. Leia algumas palavras do primeiro livro, em seguida leia algumas palavras do segundo livro e, posteriormente, do terceiro livro; então, faça um loop e leia as poucas palavras seguintes do primeiro livro e assim por diante. Depois dessa experiência, você apreciará os desafios da tecnologia multithreading — com alternância entre livros, ler brevemente, lembrar-se de onde parou em cada livro, aproximar ainda mais o livro que está lendo para poder vê-lo melhor e afastar os que não está lendo — e, no meio de todo esse caos, tentar compreender o conteúdo dos livros!

A programação de aplicativos concorrentes é um empreendimento difícil e propenso a erro. Até mesmo alguns aplicativos concorrentes mais simples estão além da capacidade de programadores iniciantes. Se achar que deve utilizar a sincronização em um programa, você deve seguir algumas diretrizes simples. Primeiro, utilize classes existentes da API do Java (como a classe `ArrayBlockingQueue`, discutida na Seção 23.9, 'Relacionamento de produtor/consumidor: `ArrayBlockingQueue`') que gerencia a sincronização para você. As classes na API do Java foram completamente testadas e depuradas e ajudam a evitar interrupções e armadilhas comuns. Segundo, se achar que precisa de funcionalidades mais personalizadas do que aquelas fornecidas nas APIs do Java, você deve utilizar a palavra-chave `synchronized` e os métodos `wait`, `notify` e `notifyAll` do `Object` (discutidos na Seção 23.12, 'Monitores e bloqueios de monitor'). Por fim, se precisar de capacidade ainda mais complexa, então você deve utilizar as `interfaces` `Lock` e `Condition`, que são introduzidas na Seção 23.5, 'Sincronização de thread').

As `interfaces` `Lock` e `Condition` são ferramentas avançadas e só devem ser utilizadas por programadores experientes que estão familiarizados com interrupções e armadilhas comuns da programação concorrente com a sincronização. Esses tópicos serão discutidos neste capítulo por várias razões. Uma delas é que eles fornecem uma base sólida para entender como os aplicativos concorrentes sincronizam o acesso à memória compartilhada. Mesmo que um aplicativo não utilize essas ferramentas explicitamente, o entendimento dos conceitos ainda é importante. Discutimos esses tópicos também para destacar os novos recursos da concorrência introduzida pelo J2SE 5.0. Por fim, mostrando a complexidade envolvida na utilização desses recursos de baixo nível, esperamos incutir em você a importância de utilizar a capacidade concorrente pré-empacotada sempre que possível.

23.2 Estados de thread: Classe Thread

A qualquer dado momento, diz-se que uma thread está em um dos vários **estados de thread** que são ilustrados no diagrama de estado UML na Figura 23.1. Alguns dos termos no diagrama são definidos nas seções posteriores.

Uma nova thread inicia seu ciclo de vida no estado *novo*. Ela permanece nesse estado até o programa iniciar a thread, o que a coloca no estado *executável*. Considera-se que uma thread nesse estado está executando sua tarefa.

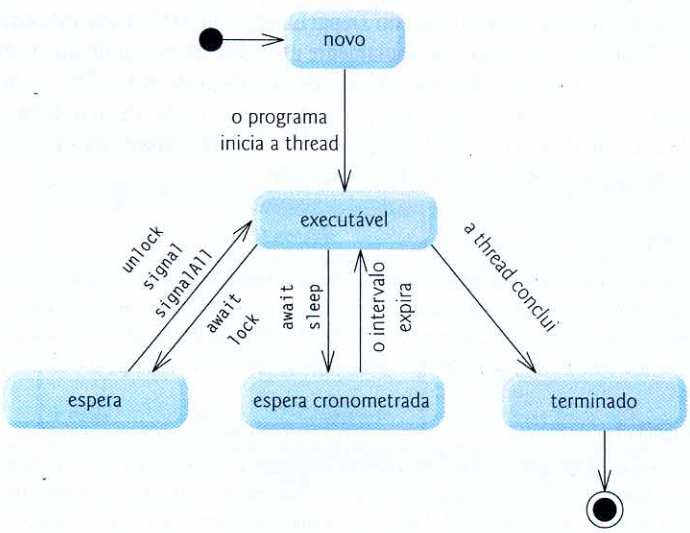


Figura 23.1 Diagrama de estado do ciclo de vida da thread.

Às vezes uma thread entra no estado de *espera* enquanto espera outra thread realizar uma tarefa. Uma vez nesse estado, a thread só volta ao estado *executável* quando outra thread sinalizar a thread de espera para retomar a execução.

Uma thread *executável* pode entrar no estado de *espera sincronizada* por um intervalo especificado de tempo. Uma thread nesse estado volta para o estado *executável* quando esse intervalo de tempo expira ou quando ocorre o evento que ele está esperando. As threads de *espera sincronizada* não podem utilizar um processador, mesmo que haja um disponível. Uma thread pode transitar para o estado de

espera sincronizada se fornecer um intervalo de espera opcional quando ela estiver esperando outra thread realizar uma tarefa. Essa thread retornará ao estado *executável* quando ela for sinalizada por outra thread ou quando o intervalo sincronizado expirar — o que ocorrer primeiro. Outra maneira de colocar uma thread no estado de *espera sincronizada* é colocá-la para dormir. Uma thread **adormecida** permanece no estado de *espera sincronizada* por um período designado de tempo (denominado **intervalo de adormecimento**) no ponto em que ele retorna para o estado *executável*. As threads dormem quando, por um breve período, não têm de realizar nenhuma tarefa. Por exemplo, um processador de texto pode conter uma thread que grave periodicamente uma cópia do documento atual no disco para fins de recuperação. Se a thread não dormisse entre os sucessivos backups, seria necessário um loop em que testaria continuamente se ela deve ou não gravar uma cópia do documento em disco. Esse loop consumiria tempo de processador sem realizar trabalho produtivo, reduzindo assim o desempenho de sistema. Nesse caso, é mais eficiente para a thread especificar um intervalo de adormecimento (igual ao período entre backups sucessivos) e entrar no estado de *espera sincronizada*. Essa thread retorna ao estado *executável* quando seu intervalo de adormecimento expira, ponto em que ela grava uma cópia do documento no disco e entra novamente no estado de *espera sincronizada*.

Uma thread *executável* entra no estado *terminado* quando completa sua tarefa ou, caso contrário, termina (talvez devido a uma condição de erro). No diagrama de estado UML, na Figura 23.1, o estado *terminado* é seguido pelo estado final da UML (símbolo do alvo) para indicar o fim das transições de estado.

No nível do sistema operacional, o estado *executável* do Java na realidade inclui dois estados separados (Figura 23.2). O sistema operacional oculta esses dois estados da Java Virtual Machine (JVM), que vê apenas o estado *executável*. Quando uma thread entra pela primeira vez no estado *executável* a partir do estado *novo*, a thread está no estado *pronto*. Uma thread *pronta* entra no estado de *execução* (isto é, começa a executar) quando o sistema operacional atribui a thread a um processador — também conhecido como **despachar a thread**. Na maioria dos sistemas operacionais, cada thread recebe uma pequena quantidade de tempo de processador — denominada **quantum** ou **fração de tempo** — com o qual realiza sua tarefa. Quando o quantum da thread expirar, a thread retornará ao estado *pronto* e o sistema operacional atribuirá outra thread ao processador (ver Seção 23.3). As transições entre esses estados são tratadas unicamente pelo sistema operacional. A JVM não 'vê' esses dois estados — ela simplesmente visualiza uma thread quando ela está no estado *executável* e a deixa para o sistema operacional fazer a transição de threads entre os estados *pronto* e de *execução*. O processo que utiliza um sistema operacional para decidir qual thread despachar é conhecido como **agendamento de thread** e depende das prioridades de thread (discutidas na próxima seção).

23.3 Prioridades de thread e agendamento de thread

Cada thread Java tem uma **prioridade** que ajuda o sistema operacional a determinar a ordem em que as threads são agendadas. As prioridades do Java estão no intervalo entre `MIN_PRIORITY` (uma constante de 1) e `MAX_PRIORITY` (uma constante de 10). Informalmente, as threads com prioridade mais alta são mais importantes para um programa e devem ser alocadas em tempo de processador antes das threads de prioridade mais baixa. Entretanto, as prioridades de thread não podem garantir a ordem em que elas são executadas. Por padrão, cada thread recebe a prioridade `NORM_PRIORITY` (uma constante de 5). Cada thread nova herda a prioridade da thread que a criou.

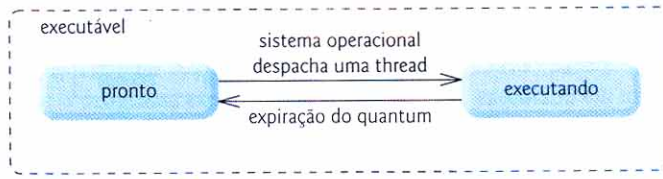


Figura 23.2 Visualização interna do sistema operacional do estado executável do Java.

[*Nota*: Essas constantes (`MAX_PRIORITY`, `MIN_PRIORITY` e `NORM_PRIORITY`) são declaradas na classe `Thread`. Recomenda-se não criar e utilizar explicitamente objetos `Thread` para implementar a concorrência, mas, em vez disso, utilizar a interface `Runnable` (descrita na Seção 23.4). A classe `Thread` contém alguns métodos `static` úteis, como veremos mais adiante neste capítulo.]

A maioria das plataformas do Java suporta o fracionamento de tempo, o que permite que threads de igual prioridade compartilhem um processador. Sem o fracionamento de tempo, cada thread em um conjunto de threads de igual prioridade executa até sua conclusão (a menos que ela deixe o estado *executável* e entre no estado de *espera* ou *espera sincronizada*, ou seja interrompida por uma thread de prioridade mais alta), antes que outras threads de igual prioridade tenham uma chance de executar. Com o fracionamento de tempo, mesmo que a thread não tenha concluído a execução quando o quantum expirar, o processador é tirado dessa thread e recebe a próxima de igual prioridade, se houver alguma disponível.

O trabalho de um **scheduler de thread** de sistema operacional é determinar a próxima thread que entra em execução. Uma simples implementação do scheduler de thread mantém a thread de prioridade mais alta *executando* o tempo todo e, se houver mais de uma thread de prioridade mais alta, isso assegura que cada uma delas executa por um quantum no estilo **rodízio**. A Figura 23.3 ilustra uma fila de múltiplos níveis de prioridade das threads. Supondo um computador de um único processador, as threads *A* e *B* executam por um quantum no esquema de rodízio até que ambas completem a execução. Isso significa que *A* obtém um quantum de tempo a executar. Então *B* obtém um quantum. Então *A* obtém outro quantum. Então *B* obtém outro quantum. Isso continua até que uma thread complete. O processador então dedica toda sua energia à thread que resta (a menos que outra thread dessa prioridade torne-se pronta). Em seguida, a thread *C* executa até

sua conclusão (considerando que nenhuma thread de prioridade mais alta chegará). Cada uma das threads *D*, *E* e *F* executa por um quantum no esquema rodízio até que todas completem a execução (novamente supondo que não há nenhuma thread de prioridade mais alta). Esse processo continua até que todas as threads executem até sua conclusão.

Dica de portabilidade 23.2

O agendamento de thread é dependente de plataforma — um aplicativo que utiliza multithreading poderia comportar-se diferentemente em implementações separadas do Java.

Quando uma thread de prioridade mais alta entra no estado *pronto*, o sistema operacional geralmente faz preempção da thread atualmente *em execução* (uma operação conhecida como **agendamento preemptivo**). Dependendo do sistema operacional, as threads de prioridade mais alta poderiam adiar —possivelmente por um tempo indefinido — a execução de threads de prioridade mais baixa. Esse **adiamento indefinido** é freqüentemente mencionado, mais alegoricamente, como **inanição**.

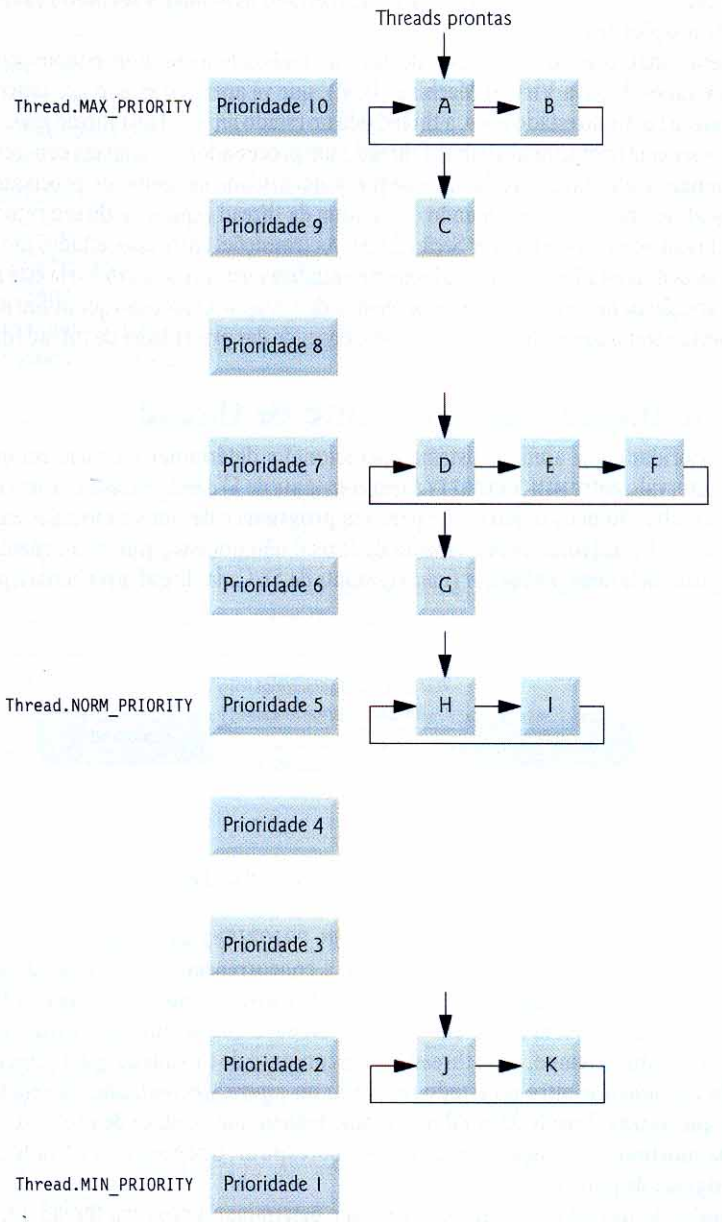


Figura 23.3 Agendamento de prioridade de threads.

Dica de portabilidade 23.3

Ao projetar applets e aplicativos que utilizam threads, você deve considerar as capacidades de threading de todas as plataformas em que os applets e aplicativos serão executados.

O J2SE 5.0 fornece utilitários de concorrência de nível mais alto para ocultar alguma complexidade e torna programas de múltiplas threads menos propensos a erros (embora eles ainda sejam certamente complexos). As prioridades de thread são ainda utilizadas nos bastidores para interagir com o sistema operacional, mas a maioria dos programadores que utiliza o multithreading do J2SE 5.0 não se preocupará com a configuração e o ajuste de prioridades de thread. Você pode aprender mais sobre as prioridades e threading em `java.sun.com/em2se/5.0/docs/api/java/lang/Thread.html`.

23.4 Criando e executando threads

Em J2SE 5.0, o modo preferido de criar um aplicativo de múltiplas threads é implementar a interface **`Runnable`** (pacote `java.lang`) e utilizar classes e métodos predefinidos para criar `Threads` que executam os objetos `Runnables`. A interface `Runnable` declara um único método chamado **`run`**. `Runnables` são executados por um objeto de uma classe que implementa a interface **`Executor`** (pacote **`java.util.concurrent`**). Essa interface declara um único método chamado **`execute`**. Em geral, um objeto `Executor` cria e gerencia um grupo de threads denominado **pool de threads**. Essas threads executam os objetos `Runnables` passados para o método `execute`. O `Executor` atribui cada `Runnable` a uma das threads disponíveis no pool de threads. Se não houver nenhuma thread disponível no pool de threads, o `Executor` cria uma nova thread ou espera que uma se torne disponível para atribui a ela o `Runnable` que foi passado para método `execute`. Dependendo do tipo `Executor`, há um limite para o número de threads que podem ser criadas. A interface **`ExecutorService`** (pacote `java.util.concurrent`) é uma subinterface de `Executor` que declara vários outros métodos para gerenciar o ciclo de vida do `Executor`. Um objeto que implementa a interface `ExecutorService` pode ser criado utilizando métodos `static` declarados na classe `Executors` (pacote `java.util.concurrent`). Utilizamos essas interfaces e métodos no próximo aplicativo, que executa três threads.

A classe `PrintTask` (Figura 23.4) implementa `Runnable` (linha 5), de modo que todo objeto `PrintTask` pode executar concorrentemente. A variável `sleepTime` (linha 7) armazena um valor inteiro aleatório (linha 17) escolhido quando o construtor `PrintTask` executa. Cada thread que executa um objeto `PrintTask` dorme pela quantidade de tempo especificada pelo objeto `PrintTask` `sleepTime` correspondente, e depois envia seu nome para a saída.

Quando uma `PrintTask` é atribuída a um processador pela primeira vez, seu método `run` (linhas 21–38) começa a execução. As linhas 25–26 exibem uma mensagem que indica o nome da thread que está executando atualmente e declara que a thread vai dormir determinado número de milissegundos. A linha 26 utiliza o campo `threadName` que foi inicializado na linha 14 com o argumento do construtor `PrintTask`. A linha 28 invoca o método `static` **`sleep`** da classe `Thread` para colocar a thread no estado de *espera sincronizada*. Nesse ponto, a thread perde o processador e o sistema permite que outra thread execute. Quando a thread acordar, ela entra novamente no estado *executável*. Quando `PrintTask` é novamente atribuída a um processador, a linha 37 gera saída do nome da thread em uma mensagem que indica que a thread não está mais dormindo — então o método `run` termina. Observe que `catch` (nas linhas 31–34) é necessário porque o método `sleep` poderia lançar uma **`InterruptedException`**, que é uma exceção verificada. Uma exceção como essa ocorre se o método **`interrupt`** de uma thread adormecida for chamado. Como a maioria dos programadores não manipula diretamente os objetos `Thread`, a ocorrência de `InterruptedExceptions` é improvável.

```
1  // Fig. 23.4: PrintTask.java
2  // Classe PrintTask dorme por um tempo aleatório de 0 a 5 segundos
3  import java.util.Random;
4
5  class PrintTask implements Runnable
6  {
7     private int sleepTime; // tempo de adormecimento aleatório para a thread
8     private String threadName; // nome da thread
9     private static Random generator = new Random();
10
11    // atribui nome à thread
12    public PrintTask( String name )
13    {
14       threadName = name; // configura nome da thread
15
16       // seleciona tempo de adormecimento aleatório entre 0 e 5 segundos
17       sleepTime = generator.nextInt( 5000 );
```

Figura 23.4 Adormecimento e despertar de threads. (Parte 1 de 2.)

```
18    } // fim do construtor PrintTask
19
20    // método run é o código a ser executado pela nova thread
21    public void run()
22    {
23       try // coloca a thread para dormir pela quantidade de tempo sleepTime
24       {
25          System.out.printf( "%s going to sleep for %d milliseconds.\n",
26             threadName, sleepTime );
27
28          Thread.sleep( sleepTime ); // coloca a thread para dormir
29       } // fim do try
30       // se a thread foi interrompida enquanto dormia, imprime o rastreamento de pilha
31       catch ( InterruptedException exception )
32       {
33          exception.printStackTrace();
34       } // fim do catch
35
36       // imprime o nome da thread
37       System.out.printf( "%s done sleeping\n", threadName );
38    } // fim do método run
39 } // fim da classe PrintTask
```

Figura 23.4 Adormecimento e despertar de threads. (Parte 2 de 2.)

A Figura 23.5 cria três threads de execução utilizando a classe `PrintTask`. O método `main` (linhas 8–28) cria e nomeia três objetos `PrintTask` (linhas 11–13). A linha 18 cria um novo `ExecutorService`. Essa linha utiliza o método **`newFixedThreadPool`** da classe `Executors`, que cria um pool consistindo em um número fixo de `Threads` como indicado pelo argumento do método (nesse caso, 3). Essas `Threads` são utilizadas pelo `threadExecutor` para executar os `Runnables`. Se o método `execute` for chamado e todas as threads em `ExecutorService` estiverem em uso, `Runnable` será colocado em uma fila e atribuído à primeira thread que completar sua tarefa anterior. O método `Executors` **`newCachedThreadPool`** retorna um `ExecutorService` que cria novas threads à medida que o aplicativo precisa delas.

As linhas 21–23 invocam o método `execute` de `ExecutorService`. Esse método cria uma nova `Thread` dentro de `ExecutorService` para executar o `Runnable` passado para ele como um argumento (nesse caso um `PrintTask`) e faz uma transição dessa `Thread` do estado *novo* para o estado *executável*. O método `execute` retorna imediatamente de cada invocação — o programa não espera cada `PrintTask` terminar. A linha 25 chama o método `ExecutorService` **`shutdown`**, que acabará cada `Thread` em `threadExecutor` assim que cada que uma concluir a execução de seu `Runnable`. A linha 27 gera a saída de uma mensagem que indica que as threads foram iniciadas. [*Nota*: A linha 18 cria o método `ExecutorService` que utiliza `newFixedThreadPool` e o argumento 3. Esse programa só executa três `Runnables`, então uma nova `Thread` será criada pelo `ExecutorService` para cada `Runnable`. Se o programa executasse mais de três `Runnables`, `Threads` adicionais não seriam criadas, mas, em vez disso, uma thread existente seria reutilizada quando concluísse o `Runnable` atribuído a ela.]

```
1  // Fig. 23.5: RunnableTester.java
2  // Impressão de múltiplas threads em diferentes intervalos.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5
6  public class RunnableTester
7  {
8     public static void main( String[] args )
9     {
10       // cria e nomeia cada executável
11       PrintTask task1 = new PrintTask( "thread1" );
12       PrintTask task2 = new PrintTask( "thread2" );
13       PrintTask task3 = new PrintTask( "thread3" );
14
```

Figura 23.5 Criação e execução de três `PrintTasks`. (Parte 1 de 2.)

```
15         System.out.println( "Starting threads" );
16
17         // cria ExecutorService para gerenciar threads
18         ExecutorService threadExecutor = Executors.newFixedThreadPool( 3 );
19
20         // inicia threads e as coloca no estado executável
21         threadExecutor.execute( task1 ); // inicia task1
22         threadExecutor.execute( task2 ); // inicia task2
23         threadExecutor.execute( task3 ); // inicia task3
24
25         threadExecutor.shutdown(); // encerra as threads trabalhadoras
26
27         System.out.println( "Threads started, main ends\n" );
28      } // fim do main
29   } // fim da classe RunnableTester
```

```
Starting threads
Threads started, main ends

thread1 going to sleep for 1217 milliseconds
thread2 going to sleep for 3989 milliseconds
thread3 going to sleep for 662 milliseconds
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

```
Starting threads
thread1 going to sleep for 314 milliseconds
thread2 going to sleep for 1990 milliseconds
Threads started, main ends

thread3 going to sleep for 3016 milliseconds
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
```

Figura 23.5 Criação e execução de três `PrintTasks`. (Parte 2 de 2.)

O código no método `main` executa na **thread principal**. Essa thread é criada pela JVM e executa o método `main`. O código no método `run` de `PrintTask` (linhas 21–38 da Figura 23.4) executa nas threads criadas pelo `ExecutorService`. Quando o método `main` termina (linha 28), o próprio programa continua executando, uma vez que ainda há threads ativas (isto é, as threads iniciadas por `threadExecutor` que ainda não alcançaram o estado *terminado*). O programa não terminará até que sua última thread complete a execução.

As saídas de exemplo desse programa mostram o nome e o tempo de adormecimento de cada thread quando ela vai dormir. A thread com o tempo de adormecimento mais curto, que normalmente acorda primeiro, indica que ela não está mais dormindo e termina. Na Seção 23.8, discutimos as razões relativas a multithreading que poderiam impedir que a thread com o tempo mais curto de adormecimento acordasse primeiro. Na primeira saída, a thread principal termina antes de qualquer outra thread gerar saída de seus nomes e tempos de adormecimento. Isso mostra que a thread principal executa até conclusão antes que qualquer uma das outras threads tenha chance de executar. Na segunda saída, as duas primeiras threads geram a saída de seus nomes e períodos de adormecimento antes de a thread principal terminar. Isso mostra que o sistema operacional permitiu a execução de outras threads antes que a thread principal terminar. Esse é um exemplo do agendamento de rodízio discutido na Seção 23.3.

23.5 Sincronização de thread

Freqüentemente, múltiplas threads de execução manipulam um objeto compartilhado na memória. Quando isso ocorre e esse objeto é modificado por uma ou mais threads, podem ocorrer resultados indeterminados (como veremos nos exemplos do capítulo), a menos que o objeto compartilhado seja gerenciado adequadamente. Se uma thread estiver no processo de atualização de um objeto compartilhado e outra também tentar chamá-lo, é possível que parte do objeto reflita as informações de uma thread enquanto outra parte do objeto reflita informações de uma thread diferente. Quando isso acontece, o comportamento do programa não pode ser confiável. Às vezes o programa produzirá tanto resultados corretos como resultados incorretos. Em qualquer caso não há nenhuma mensagem de erro que indique que o objeto compartilhado foi manipulado incorretamente.

O problema pode ser resolvido fornecendo a uma thread por vez o código de acesso exclusivo que manipula o objeto compartilhado. Durante esse tempo, as outras threads que desejam manipular o objeto são mantidas na espera. Quando a thread com acesso exclusivo ao objeto terminar de chamá-lo, uma das threads que foi mantida na espera tem a permissão de prosseguir. Desse modo, toda thread que acessa o objeto compartilhado exclui todas as outras threads de fazer isso simultaneamente. Isso é chamado de **exclusão mútua**. A exclusão mútua permite ao programador realizar a **sincronização de threads**, que coordena o acesso aos dados compartilhados por múltiplas threads concorrentes.

O Java utiliza **bloqueios** para realizar a sincronização. Qualquer objeto pode conter um objeto que implementa a interface `Lock` (pacote `java.util.concurrent.locks`). Uma thread chama o método `lock` de `Lock` para **obter o bloqueio**. Uma vez que um `Lock` foi obtido por uma thread, o objeto `Lock` não permitirá que outra thread obtenha o bloqueio até que a primeira thread libere o `Lock` (chamando o método **`unlock`** de `Lock`). Se há várias threads tentando chamar o método `lock` no mesmo objeto `Lock` ao mesmo tempo, somente uma thread pode obter o bloqueio por vez — as outras threads que tentarem obter o `Lock` contido no mesmo objeto serão colocadas no estado de *espera* desse bloqueio. Quando uma thread chamar o método `unlock`, o bloqueio no objeto será liberado e a thread na espera de prioridade mais alta que tentar bloquear o objeto prosseguirá. A classe **`ReentrantLock`** (pacote `java.util.concurrent.locks`) é uma implementação básica da interface `Lock`. O construtor de um `ReentrantLock` aceita um argumento `boolean` que especifica se o bloqueio tem uma **diretiva de imparcialidade**. Se isso estiver configurado como `true`, a diretiva de imparcialidade do `ReentrantLock` determina que a thread na espera mais longa vai obter o bloqueio quando ela estiver disponível. Se este estiver configurado como `false`, não é garantido quanto a que thread na espera vai obter o bloqueio quando ela estiver disponível.

Dica de desempenho 23.2

Utilizar um `Lock` com uma diretiva de imparcialidade impede o adiamento indefinido, mas também pode reduzir drasticamente a eficiência total de um programa. Devido à grande queda no desempenho, os bloqueios imparciais são necessários apenas em circunstâncias extremas.

Se uma thread que possui o bloqueio em um objeto determinar que não é possível continuar sua tarefa até que alguma condição seja satisfeita, a thread pode esperar em uma **variável de condição**. Isso dispensa a thread da disputa pelo processador, coloca-a em uma fila de espera pela variável de condição e libera o bloqueio no objeto. As variáveis de condição devem ser associadas com um `Lock` e são criadas chamando o método `Lock` **`newCondition`**, que retorna um objeto que implementa a interface **`Condition`** (pacote `java.util.concurrent.locks`). Para esperar uma variável de condição, a thread pode chamar o método **`await`** de `Condition`. Isso libera imediatamente o `Lock` associado e coloca a thread no estado de *espera* dessa `Condition`. Outras threads podem então tentar obter o `Lock`. Quando uma thread *executável* completar uma tarefa e determinar que a thread na *espera* pode agora continuar, a thread *executável* pode chamar o método `Condition` **`signal`** para permitir que uma thread no estado de *espera* dessa `Condition` retorne ao estado *executável*. Nesse ponto, a thread que fez a transição do estado de *espera* para o estado *executável* pode tentar readquirir o `Lock` no objeto. Mesmo se fosse capaz de readquirir o `Lock`, a thread ainda poderia não ser capaz de realizar sua tarefa nesse momento — caso em que pode chamar o método `await` para liberar o `Lock` e entrar novamente no estado de *espera*. Se múltiplas threads estiverem no estado de *espera* de uma `Condition` quando `signal` for chamado, a implementação padrão de `Condition` sinaliza a thread de espera mais longa para mudar para o estado *executável*. Se uma thread chamar o método `Condition` **`signalAll`**, todas as threads que esperam essa condição mudam para o estado *executável* e tornam-se elegíveis para readquirir o `Lock`. Apenas uma dessas threads pode obter o `Lock` no objeto por vez — outras threads que tentarem adquirir o mesmo `Lock` esperarão até que o `Lock` se torne disponível novamente. Se o `Lock` foi criado com uma diretiva de imparcialidade, a thread de espera mais longa então adquirirá o `Lock`. Quando uma thread concluir sua tarefa com um objeto compartilhado, ela deve chamar o método `unlock` para liberar o `Lock`.

Erro comum de programação 23.1

O impasse (deadlock) ocorre quando uma thread em espera (vamos chamá-la de thread1) não pode prosseguir porque está esperando (direta ou indiretamente) outra thread (vamos chamá-la de thread2) prosseguir; simultaneamente, a thread2 não pode prosseguir porque está esperando (direta ou indiretamente) a thread1 prosseguir. Como duas threads estão esperando uma à outra, as ações que permitiriam a cada uma continuar a execução nunca ocorrem.

Dica de prevenção de erros 23.1

Quando múltiplas threads manipulam um objeto compartilhado utilizando bloqueios, assegure que, se uma thread chamar o método `await` para entrar no estado de espera por uma variável de condição, uma thread separada por fim chame o método `Condition` `signal` para fazer a transição da thread em espera pela variável de condição de volta para o estado executável. Se múltiplas threads podem estar esperando a variável de condição, uma thread separada pode chamar o método `Condition` `signalAll` como uma salvaguarda para assegurar que todas as threads na espera tenham outra oportunidade de realizar suas tarefas. Se isso não for feito, o adiamento indefinido ou impasse poderia ocorrer.

Observação de engenharia de software 23.1

O bloqueio que ocorre com a execução dos métodos `lock` e `unlock` poderia levar a um impasse se os bloqueios nunca fossem liberados. As chamadas para o método `unlock` devem ser colocadas em blocos `finally` para assegurar que os bloqueios sejam liberados e evitar esses tipos de impasses.

Dica de desempenho 23.3

A sincronização para alcançar a precisão em programas de múltiplas threads pode tornar a execução de programas mais lenta, como resultado de overhead de thread e da transição freqüente de threads entre os estados de espera e executável. Não há, entretanto, muito a dizer sobre programas multiencadeados altamente eficientes, mas incorretos!

Erro comum de programação 23.2

É um erro se uma thread emitir um `await`, um `signal` ou um `signalAll` em uma variável de condição sem adquirir o bloqueio dessa variável de condição. Isso causa uma **IllegalMonitorStateException**.

23.6 Relacionamento entre produtor e consumidor sem sincronização

Em um **relacionamento produtor/consumidor**, a parte **produtora** de um aplicativo gera dados e os armazena em um objeto compartilhado, e a parte **consumidora** de um aplicativo lê os dados do objeto compartilhado. Um exemplo de relacionamento produtor/consumidor comum é o spooling de impressão. Um processador de texto faz um spool de dados para um buffer (em geral um arquivo) e esses dados são subseqüentemente consumidos pela impressora à medida que ela imprime o documento. De maneira semelhante, um aplicativo que copia dados em discos a laser coloca os dados em um buffer de tamanho fixo que é esvaziado quando a unidade CD-RW grava os dados no disco a laser.

Em um relacionamento produtor/consumidor de múltiplas threads, uma **thread produtora** gera dados e os coloca em um objeto compartilhado chamado **buffer**. Uma **thread consumidora** lê dados do buffer. Se a produtora que está esperando para colocar os próximos dados no buffer determinar que a consumidora ainda não leu os dados anteriores, a thread produtora deve chamar `await`, de modo que o consumidor possa ler os dados antes das atualizações adicionais — caso contrário, a consumidora nunca vê os dados anteriores e estes são perdidos para o aplicativo. Quando a thread consumidora ler os dados, ela deve chamar `signal` para permitir que uma produtora em espera armazene o próximo valor. Se uma thread consumidora localizar o buffer vazio ou achar que os dados anteriores já foram lidos, a consumidora deve chamar `await` — caso contrário a consumidora poderia ler dados antigos outra vez a partir do buffer. Quando a produtora colocar os próximos dados no buffer, a produtora deve chamar `signal` para permitir que a thread consumidora prossiga, de modo que a consumidora possa ler os novos dados.

Vamos considerar como os erros de lógica podem surgir se não sincronizarmos o acesso entre múltiplas threads que manipulam dados compartilhados. Nosso próximo exemplo (figuras 23.6 e 23.10) implementa um relacionamento produtor/consumidor em que uma thread de produtor grava os números de 1 a 10 em um buffer compartilhado — uma posição da memória compartilhada entre duas threads (uma variável `int` única chamada `buffer` na linha 6 da Figura 23.9 nesse exemplo). A thread consumidora lê esses dados do buffer compartilhado e os exibe. A saída do programa mostra os valores que a produtora grava (produz) no buffer compartilhado e os valores que a consumidora lê (consome) a partir do buffer compartilhado.

Cada valor que a thread produtora gravar no buffer compartilhado deve ser consumido exatamente uma vez pela thread consumidora. Entretanto, as threads nesse exemplo não são sincronizadas. Portanto, os dados podem ser perdidos se a produtora colocar novos dados no buffer compartilhado antes de a consumidora consumir os dados anteriores. Além disso, os dados podem ser duplicados incorretamente se a consumidora consumir os dados outra vez antes de a produtora produzir o próximo valor. Para mostrar essas possibilidades, a thread consumidora no exemplo a seguir mantém um total de todos os valores que ela lê. A thread produtora produz valores de 1 a 10. Se a consumidora ler cada valor produzido uma vez e apenas uma vez, o total será 55. Entretanto, se executar esse programa várias vezes, você verá que o total nem sempre é 55 (como mostrado nas saídas da Figura 23.10). Para enfatizar a questão, as threads produtoras e consumidoras no exemplo dormem por intervalos aleatórios de até três segundos entre a execução de suas tarefas. Portanto, não sabemos exatamente quando a thread produtora tentará gravar um novo valor, nem quando a thread consumidora tentará ler um valor.

O programa consiste na interface `Buffer` (Figura 23.6) e em quatro classes — `Producer` (Figura 23.7), `Consumer` (Figura 23.8), `UnsynchronizedBuffer` (Figura 23.9) e `SharedBufferTest` (Figura 23.10). A interface `Buffer` declara os métodos `set` e `get` que um `Buffer` deve implementar para permitir à thread `Producer` colocar um valor no `Buffer` e à thread `Consumer` recuperar um valor dele. Veremos a implementação dessa interface na Figura 23.9.

A classe `Producer` (Figura 23.7) implementa a interface `Runnable`, permitindo que ela seja executada em uma thread separada. O construtor (linhas 11–14) inicializa a referência `Buffer` `sharedLocation` com um objeto criado em `main` (linha 14 da Figura 23.10) e passado para o construtor no parâmetro `shared`. Como veremos, esse é um objeto `UnsynchronizedBuffer` que implementa a interface `Buffer` sem sincronizar o acesso ao objeto compartilhado. A thread `Producer` nesse programa executa as tarefas especificadas no método `run` (linhas 17–39). Cada iteração do loop (linhas 21–35) invoca o método `Thread` `sleep` (linha 25) para colocar a thread `Producer` no estado de *espera sincronizada* por um intervalo aleatório de tempo entre 0 e 3 segundos. Quando a thread acordar, a linha 26 passa o valor da variável de controle `count` para o método `set` do objeto `Buffer` a fim de configurar o valor do buffer compartilhado. A linha 27 mantém um total de todos os valores produzidos até agora, e a linha 28 envia esse valor para a saída. Quando o loop for concluído, as linhas 37–38 exibem uma mensagem que indica que a thread conclui a produção de dados e está terminando. Em seguida, o método `run` termina, indicando que `Producer` completou sua tarefa. É importante observar que qualquer método chamado a partir do método `run` de uma thread (por exemplo, o método `Buffer` `set`) executa como parte dessa thread de execução. De fato, cada thread tem sua própria pilha de chamadas de método. Esse fato torna-se importante na Seção 23.7 quando adicionamos a sincronização ao relacionamento produtor/consumidor.

A classe `Consumer` (Figura 23.8) também implementa a interface `Runnable`, permitindo que `Consumer` seja executada concorrentemente com `Producer`. O construtor (linhas 11–14) inicializa a referência `Buffer sharedLocation` com um objeto que implementa a interface `Buffer` criada em `main` (Figura 23.10) e passada para o construtor como o parâmetro `shared`. Como veremos, esse é o mesmo objeto `UnsynchronizedBuffer` que é utilizado para inicializar o objeto `Producer` — portanto, as duas threads compartilham o mesmo objeto.

```
1  // Fig. 23.6: Buffer.java
2  // Interface Buffer especifica métodos chamados por Producer e Consumer.
3
4  public interface Buffer
5  {
6     public void set( int value ); // coloca o valor int no Buffer
7     public int get(); // retorna o valor int a partir do Buffer
8  } // fim da interface Buffer
```

Figura 23.6 Interface `Buffer` utilizada nos exemplos de produtor/consumidor.

```
1  // Fig. 23.7: Producer.java
2  // O método run do Producer armazena os valores de 1 a 10 no buffer.
3  import java.util.Random;
4
5  public class Producer implements Runnable
6  {
7     private static Random generator = new Random();
8     private Buffer sharedLocation; // referência a objeto compartilhado
9
10    // construtor
11    public Producer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // fim do construtor Producer
15
16    // armazena os valores de 1 a 10 em sharedLocation
17    public void run()
18    {
19       int sum = 0;
20
21       for ( int count = 1; count <= 10; count++ )
22       {
23          try // dorme de 0 a 3 segundos, então coloca valor em Buffer
24          {
25             Thread.sleep( generator.nextInt( 3000 ) ); // a thread dorme
26             sharedLocation.set( count ); // configura valor no buffer
27             sum += count; // incrementa soma de valores
28             System.out.printf( "\t%2d\n", sum );
29          } // fim do try
30          // se a thread adormecida é interrompida, imprime rastreamento de pilha
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // fim do catch
35       } // fim do for
36
37       System.out.printf( "\n%s\n%s\n", "Producer done producing.",
38          "Terminating Producer." );
39    } // fim do método run
40 } // fim da classe Producer
```

Figura 23.7 `Producer` representa a thread produtora em um relacionamento produtor/consumidor.

A thread `Consumer` nesse programa realiza as tarefas especificadas no método `run` (linhas 17–39). O loop nas linhas 21–35 itera dez vezes. Cada iteração do loop invoca o método `Threadsleep` (linha 26), que coloca a thread `Consumer` no estado de *espera sincronizada* entre 0 e 3 segundos. Em seguida, a linha 27 utiliza o método `get` de `Buffer` para recuperar o valor no buffer compartilhado, então adiciona o valor à variável `sum`. A linha 28 exibe o total dos valores consumidos até agora. Quando o loop for concluído, as linhas 37–38 exibem uma linha com a soma dos valores consumidos. Então o método `run` termina, o que indica que `Consumer` completou sua tarefa. Depois que ambas as threads entram no estado *terminado*, o programa é encerrado.

```
1  // Fig. 23.8: Consumer.java
2  // O método run de Consumer itera dez vezes lendo um valor do buffer.
3  import java.util.Random;
4
5  public class Consumer implements Runnable
6  {
7     private static Random generator = new Random();
8     private Buffer sharedLocation; // referência a objeto compartilhado
9
10    // construtor
11    public Consumer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // fim do construtor Consumer
15
16    // lê o valor do sharedLocation quatro vezes e soma os valores
17    public void run()
18    {
19       int sum = 0;
20
21       for ( int count = 1; count <= 10; count++ )
22       {
23          // dorme de 0 a 3 segundos, lê o valor do buffer e adiciona a soma
24          try
25          {
26             Thread.sleep( generator.nextInt( 3000 ) );
27             sum += sharedLocation.get();
28             System.out.printf( "\t\t\t%2d\n", sum );
29          } // fim do try
30          // se a thread adormecida é interrompida, imprime rastreamento de pilha
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // fim do catch
35       } // fim do for
36
37       System.out.printf( "\n%s %d.\n%s\n",
38          "Consumer read values totaling", sum, "Terminating Consumer." );
39    } // fim do método run
40 } // fim da classe Consumer
```

Figura 23.8 `Consumer` representa a thread consumidora em um relacionamento produtor/consumidor.

[*Nota*: Utilizamos o método `sleep` no método `run` das classes `Producer` e `Consumer` para enfatizar o fato de que, em aplicativos com múltiplas threads, é imprevisível o momento em que cada thread realizará sua tarefa e por quanto tempo ela realizará a tarefa quando tiver um processador. Normalmente, essas questões de agendamento de thread são de responsabilidade do sistema operacional do computador e, portanto, estão além do controle do desenvolvedor Java. Nesse programa, as tarefas de nossa thread são bem simples — para `Producer`, gravar os valores de 1 a 10 no buffer e, para `Consumer`, ler 10 valores do buffer e adicionar cada valor à variável `sum`. Sem a chamada de método `sleep` e se `Producer` executasse primeiro, considerando os processadores fenomenalmente rápidos de hoje, `Producer` possivelmente completaria sua tarefa antes de `Consumer` ter uma chance de executar. Se `Consumer` executasse primeiro, ela possivelmente consumiria –1 dez vezes e terminaria antes que `Producer` pudesse produzir o primeiro valor real.]

A classe `UnsynchronizedBuffer` (Figura 23.9) implementa a interface `Buffer` (linha 4). Um objeto dessa classe é compartilhado entre `Producer` e `Consumer`. A linha 6 declara a variável de instância `buffer` e a inicializa com o valor –1. Esse valor é utilizado para demonstrar o caso em que `Consumer` tenta consumir um valor antes que `Producer` coloque algum valor em `buffer`. Os métodos `set` (linhas 9–13) e `get` (linhas 16–20) não sincronizam o acesso ao campo `buffer`. O método `set` simplesmente atribui seu argumento a `buffer` (linha 12) e o método `get` simplesmente retorna o valor de `buffer` (linha 19).

A classe `SharedBufferTest` contém o método `main` (linhas 8–32), que carrega o aplicativo. A linha 11 cria um `ExecutorService` com duas threads — uma para executar `Producer` e a outra para executar `Consumer`. A linha 14 cria um objeto `UnsynchronizedBuffer` e atribui sua referência à variável `Buffer` `sharedLocation`. Esse objeto armazena os dados que serão compartilhados entre as threads `Producer` e `Consumer`. As linhas 23–24 criam e executam `Producer` e `Consumer`. Observe que os construtores `Producer` e `Consumer` recebem o mesmo objeto `Buffer` (`sharedLocation`), assim cada objeto é inicializado com uma referência ao mesmo `Buffer`. Essas linhas também carregam implicitamente as threads e chamam o método `run` de cada `Runnable`. Por fim, a linha 31 chama o método `shutdown`, de modo que o aplicativo possa terminar quando as threads `Producer` e `Consumer` completarem suas tarefas. Quando o método `main` terminar (linha 32), a thread principal de execução entra no estado *terminado*.

Considerando a visão geral desse exemplo, lembre-se de que gostaríamos que a thread `Producer` executasse primeiro e que cada valor produzido pelo `Producer` fosse consumido exatamente uma vez pelo `Consumer`. Entretanto, quando estudamos a primeira saída da Figura 23.10, vemos que `Producer` grava um valor três vezes, antes que `Consumer` leia seu primeiro valor (3). Portanto, os valores 1 e 2 são perdidos. Posteriormente, os valores 5, 6 e 9 são perdidos, enquanto 7 e 8 são lidos duas vezes e 10 é lido quatro vezes. Então, a primeira saída produziu um total incorreto de 77, em vez de um total correto de 55. Na segunda saída, observe que o `Consumer` lê antes de o `Producer` ter gravado algum valor. Observe também que `Consumer` já leu cinco vezes antes de `Producer` gravar o valor 2. Entretanto, os valores 5, 7, 8, 9 e 10 são inteiramente perdidos. Uma saída incorreta de 19 é produzida. (As linhas na saída em que `Producer` ou `Consumer` atuou fora de ordem estão destacadas.) Esse exemplo demonstra claramente que o acesso a um objeto compartilhado por threads concorrentes deve ser controlado cuidadosamente, pois, do contrário, um programa pode produzir resultados incorretos.

```
1  // Fig. 23.9: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer representa um único inteiro compartilhado.
3
4  public class UnsynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // compartilhado pelas threads producer e consumer
7
8     // coloca o valor no buffer
9     public void set( int value )
10    {
11       System.out.printf( "Producer writes\t%2d", value );
12       buffer = value;
13    } // fim do método set
14
15    // retorna valor do buffer
16    public int get()
17    {
18       System.out.printf( "Consumer reads\t%2d", buffer );
19       return buffer;
20    } // fim do método get
21 } // fim da classe UnsynchronizedBuffer
```

Figura 23.9 `UnsynchronizedBuffer` mantém o inteiro compartilhado que é acessado por uma thread produtora e uma consumidora por meio dos métodos `set` e `get`.

Para resolver os problemas de dados perdidos e duplicados, a Seção 23.7 apresenta um exemplo em que utilizamos um `Lock` e os métodos `Condition` `await` e `signal` para sincronizar o acesso ao código que manipula o objeto compartilhado, garantindo que todos os valores serão processados uma vez e apenas uma vez. Quando uma thread adquire um bloqueio, nenhuma outra thread pode adquirir esse mesmo bloqueio até a thread original liberá-lo.

```
1  // Fig. 23.10: SharedBufferTest.java
2  // Aplicativo mostra duas threads que manipulam um buffer não-sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
```

Figura 23.10 `SharedBufferTest` configura um aplicativo produtor/consumidor que utiliza um buffer não-sincronizado. (Parte 1 de 3.)

```
5
6  public class SharedBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // cria UnsynchronizedBuffer para armazenar ints
14       Buffer sharedLocation = new UnsynchronizedBuffer();
15
16       System.out.println( "Action\t\tValue\tProduced\tConsumed" );
17       System.out.println( "------\t\t-----\t--------\t--------\n" );
18
19       // tenta iniciar as threads produtora e consumidora fornecendo acesso a cada uma
20       // a sharedLocation
21       try
22       {
23          application.execute( new Producer( sharedLocation ) );
24          application.execute( new Consumer( sharedLocation ) );
25       } // fim do try
26       catch ( Exception exception )
27       {
28          exception.printStackTrace();
29       } // fim do catch
30
31       application.shutdown(); // termina aplicativo quando as threads terminam
32    } // fim do main
33 } // fim da classe SharedBufferTest
```

```
Action          Value   Produced        Consumed
------          -----   ---------       ---------

Producer writes  1       1
Producer writes  2       3
Producer writes  3       6
Consumer reads   3                       3
Producer writes  4      10
Consumer reads   4                       7
Producer writes  5      15
Producer writes  6      21
Producer writes  7      28
Consumer reads   7                      14
Consumer reads   7                      21
Producer writes  8      36
Consumer reads   8                      29
Consumer reads   8                      37
Producer writes  9      45
Producer writes 10      55

Producer done producing.
Terminating Producer.
Consumer reads  10                      47
Consumer reads  10                      57
Consumer reads  10                      67
Consumer reads  10                      77

Consumer read values totaling 77.
Terminating Consumer.
```

Figura 23.10 SharedBufferTest configura um aplicativo produtor/consumidor que utiliza um buffer não-sincronizado. (Parte 2 de 3.)

```
Action          Value   Produced        Consumed
------          ------  ---------       ---------

Consumer reads  -1                      -1
Producer writes  1       1
Consumer reads   1                       0
Consumer reads   1                       1
Consumer reads   1                       2
Consumer reads   1                       3
Consumer reads   1                       4
Producer writes  2       3
Consumer reads   2                       6
Producer writes  3       6
Consumer reads   3                       9
Producer writes  4      10
Consumer reads   4                      13
Producer writes  5      15
Producer writes  6      21
Consumer reads   6                      19

Consumer read values totaling 19.
Terminating Consumer.
Producer writes  7      28
Producer writes  8      36
Producer writes  9      45
Producer writes 10      55

Producer done producing.
Terminating Producer.
```

Figura 23.10 SharedBufferTest configura um aplicativo produtor/consumidor que utiliza um buffer não-sincronizado. (Parte 2 de 3.)

23.7 Relacionamento entre produtor e consumidor com sincronização

O aplicativo nas figuras 23.11 e 23.12 mostra uma produtora e uma consumidora que acessam um buffer compartilhado com sincronização. Nesse caso, a consumidora consome corretamente um valor apenas depois de a produtora ter produzido um valor; esta, por sua vez, produz corretamente um novo valor apenas depois de a consumidora ter consumido o valor produzido anteriormente. Reutilizamos a interface `Buffer` (Figura 23.6) e utilizamos as classes `Producer` (Figura 23.7 — modificadas para remover a linha 28) e `Consumer` (Figura 23.8 — modificadas para remover a linha 28) do exemplo na Seção 23.6. Essa abordagem permite demonstrar que as threads que acessam o objeto compartilhado não estão cientes de que estão sendo sincronizadas. O código que realiza a sincronização é colocado nos métodos `set` e `get` da classe `SynchronizedBuffer` (Figura 23.11), que implementa a interface `Buffer` (linha 7). Portanto, os métodos `run` de `Producer` e de `Consumer` simplesmente chamam os métodos `set` e `get` do objeto compartilhado, como no exemplo da Seção 23.6.

```
1  // Fig. 23.11: SynchronizedBuffer.java
2  // SynchronizedBuffer sincroniza acesso a um único inteiro compartilhado.
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5  import java.util.concurrent.locks.Condition;
6
7  public class SynchronizedBuffer implements Buffer
8  {
9     // Bloqueio para controlar a sincronização com esse buffer
10    private Lock accessLock = new ReentrantLock();
11
12    // condições para controlar a leitura e gravação
13    private Condition canWrite = accessLock.newCondition();
14    private Condition canRead = accessLock.newCondition();
15
16    private int buffer = -1; // compartilhado pelas threads produtora e consumidora
17    private boolean occupied = false; // se o buffer estiver ocupado
```

Figura 23.11 SynchronizedBuffer sincroniza acesso a um inteiro compartilhado. (Parte 1 de 3.)

```
18
19    // coloca o valor int no buffer
20    public void set( int value )
21    {
22       accessLock.lock(); // bloqueia esse objeto
23
24       // envia informações de thread e de buffer para a saída, então espera
25       try
26       {
27          // enquanto o buffer não estiver vazio, coloca thread no estado de espera
28          while ( occupied )
29          {
30             System.out.println( "Producer tries to write." );
31             displayState( "Buffer full. Producer waits." );
32             canWrite.await(); // espera até que o buffer esteja vazio
33          } // fim do while
34
35          buffer = value; // configura novo valor de buffer
36
37          // indica que a produtora não pode armazenar outro valor
38          // até a consumidora recuperar valor atual de buffer
39          occupied = true;
40
41          displayState( "Producer writes " + buffer );
42
43          // sinaliza a thread que está esperando para ler a partir do buffer
44          canRead.signal();
45       } // fim do try
46       catch ( InterruptedException exception )
47       {
48          exception.printStackTrace();
49       } // fim do catch
50       finally
51       {
52          accessLock.unlock(); // desbloqueia esse objeto
53       } // fim de finally
54    } // fim do método set
55
56    // retorna valor do buffer
57    public int get()
58    {
59       int readValue = 0; // inicializa valor lido a partir do buffer
60       accessLock.lock(); // bloqueia esse objeto
61
62       // envia informações de thread e de buffer para a saída, então espera
63       try
64       {
65          // enquanto os dados não são lidos, coloca thread em estado de espera
66          while ( !occupied )
67          {
68             System.out.println( "Consumer tries to read." );
69             displayState( "Buffer empty. Consumer waits." );
70             canRead.await(); // espera até o buffer tornar-se cheio
71          } // fim do while
```

Figura 23.11 SynchronizedBuffer sincroniza acesso a um inteiro compartilhado. (Parte 2 de 3.)

```
72
73          // indica que a produtora pode armazenar outro valor
74          // porque a consumidora acabou de recuperar o valor do buffer
75          occupied = false;
76
77          readValue = buffer; // recupera o valor do buffer
78          displayState( "Consumer reads " + readValue );
79
80          // sinaliza a thread que está esperando o buffer tornar-se vazio
81          canWrite.signal();
82       } // fim do try
83       // se a thread na espera tiver sido interrompida, imprime o rastreamento de pilha
84       catch ( InterruptedException exception )
85       {
86          exception.printStackTrace();
87       } // fim do catch
88       finally
89       {
90          accessLock.unlock(); // desbloqueia esse objeto
91       } // fim de finally
92
93       return readValue;
94    } // fim do método get
95
96    // exibe a operação atual e o estado de buffer
97    public void displayState( String operation )
98    {
99       System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
100         occupied );
101   } // fim do método displayState
102 } // fim da classe SynchronizedBuffer
```

Figura 23.11 SynchronizedBuffer sincroniza acesso a um inteiro compartilhado. (Parte 3 de 3.)

A classe SynchronizedBuffer (Figura 23.11) contém cinco campos. A linha 10 cria um novo objeto de tipo ReentrantLock e atribui sua referência à variável Lock accessLock. ReentrantLock é criado sem a diretiva de imparcialidade porque um único Producer ou Consumer estará esperando para adquirir o Lock nesse exemplo. As linhas 13–14 criam duas Conditions utilizando o método Lock newCondition. A Condition canWrite contém uma fila de threads que esperam o buffer tornar-se cheio (isto é, ainda há dados no buffer para o Consumer ler). Se o buffer estiver cheio, Producer chama o método await nessa Condition. Quando Consumer lê os dados de um buffer cheio, chama o método signal nessa Condition. A Condition canRead contém uma fila de threads que esperam o buffer esvaziar-se (isto é, não há dados no buffer para o Consumer ler). Se o buffer estiver vazio, Consumer chama o método await nessa Condition. Quando Producer gravar no buffer vazio, ele chama o método signal nessa Condition. O int buffer (linha 16) armazena os dados compartilhados e a variável boolean occupied (linha 17) monitora se o buffer atualmente armazena ou não dados (que Consumer deve ler).

A linha 22 no método set chama o método lock do accessLock de SynchronizedBuffer. Se o bloqueio estiver disponível (isto é, nenhuma outra thread obteve esse bloqueio), o método lock retornará imediatamente (essa thread agora possui o bloqueio) e a thread prosseguirá. Se o bloqueio estiver indisponível (isto é, o bloqueio foi armazenado por outra thread), esse método esperará até que o bloqueio seja liberado. Depois que o bloqueio é obtido, o bloco try nas linhas 25–45 é executado. A linha 28 testa occupied para determinar se o buffer está cheio. Se estiver, as linhas 30–31 geram a saída que a thread esperará. A linha 32 chama o método Condition await na variável de condição canWrite que liberará temporariamente o bloqueio de SynchronizedBuffer e esperará um sinal de Consumer de que o buffer está disponível para gravação. Quando o buffer estiver disponível para gravação, o método prossegue, gravando no buffer (linha 35), configurando occupied como true (linha 39) e enviando para a saída o fato de que foi a produtora que gravou um valor. A linha 44 chama o método Condition signal na variável de condição canRead para notificar o Consumer em espera de que o buffer tem novos dados a serem lidos. A linha 52 chama o método unlock dentro de um bloco finally para liberar o bloqueio e permitir que o Consumer prossiga.

Erro comum de programação 23.3

Coloque as chamadas para o método `Lock` `unlock` em um bloco `finally`. Se uma exceção for lançada, o desbloqueio ainda deve ser chamado ou o impasse pode ocorrer.

A linha 60 do método `get` (linhas 57–94) chama o método `lock` para obter o bloqueio desse objeto. Esse método esperará até que o bloqueio esteja disponível. Uma vez que o bloqueio é obtido, a linha 66 testa se `occupied` é `false`, indicando que o buffer não tem dados. Se o buffer estiver vazio, a linha 70 chama o método `await` na variável de condição `canRead`. Lembre-se de que o método `signal` é chamado na variável `canRead` no método `set` (linha 44). Quando a variável de condição é sinalizada, o método `get` continua. A linha 75 configura `occupied` como `false`, a linha 77 armazena o valor de `buffer` em `readValue`, e a linha 78 envia o `readValue` para a saída. Então a linha 81 sinaliza a variável de condição `canWrite`. Isso despertará `Producer` se ele, de fato, estiver esperando pelo buffer a ser esvaziado. A linha 90 chama o método `unlock` em um bloco `finally` para liberar o bloqueio e a linha 93 retorna o valor do buffer para o método chamador.

Observação de engenharia de software 23.2

Sempre invoque o método `await` em um loop que testa uma condição apropriada. É possível que uma thread entre novamente no estado executável antes que a condição que ela estava esperando seja satisfeita. Testar a condição novamente assegura que a thread não executará de maneira errada se ela tiver sido sinalizada anteriormente.

Erro comum de programação 23.4

Esquecer de sinalizar (`signal`) uma thread que está esperando por uma condição é um erro de lógica. A thread permanecerá no estado de espera, o que a impedirá de continuar trabalhando. Tal espera pode levar ao adiamento indefinido ou impasse.

A classe `SharedBufferTest2` (Figura 23.12) é semelhante à classe `SharedBufferTest` (Figura 23.10). `SharedBufferTest2` contém o método `main` (linhas 8–30), que carrega o aplicativo. A linha 11 cria um `ExecutorService` com duas threads para executar `Producer` e `Consumer`. A linha 14 cria um objeto `SynchronizedBuffer` e atribui sua referência à variável `Buffer` `sharedLocation`. Esse objeto armazena os dados que serão compartilhados entre as threads `Producer` e `Consumer`. As linhas 16–17 exibem os títulos de coluna na saída. As linhas 21–22 executam `Producer` e `Consumer`. Por fim, a linha 29 chama o método `shutdown` para encerrar o aplicativo quando `Producer` e `Consumer` completarem suas tarefas. Quando o método `main` conclui (linha 30), a thread principal de execução termina.

Analise as saídas na Figura 23.12. Observe que cada inteiro produzido é consumido exatamente uma vez — nenhum valor é perdido e nenhum valor é consumido mais de uma vez. A sincronização e as variáveis de condição asseguram que `Producer` e `Consumer` não podem realizar suas tarefas, a menos que seja sua vez. `Producer` deve ir primeiro, `Consumer` deve esperar se `Producer` não tiver produzido desde que `Consumer` foi consumida pela última vez e `Producer` deve esperar se `Consumer` ainda não tiver consumido o valor que `Producer` produziu mais recentemente. Execute esse programa várias vezes para confirmar que todo inteiro produzido é consumido exatamente uma vez. Na saída de exemplo, observe as linhas que indicam quando `Producer` e `Consumer` devem esperar para realizar suas respectivas tarefas.

```
1  // Fig. 23.12: SharedBufferTest2.java
2  // Aplicativo mostra duas threads que manipulam um buffer sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // cria SynchronizedBuffer para armazenar ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t--------" );
18
19       try // tenta iniciar a produtora e a consumidora
```

Figura 23.12 `SharedBufferTest2` configura um aplicativo produtor/consumidor que utiliza um buffer sincronizado. (Parte 1 de 3.)

```
20         {
21            application.execute( new Producer( sharedLocation ) );
22            application.execute( new Consumer( sharedLocation ) );
23         } // fim do try
24         catch ( Exception exception )
25         {
26            exception.printStackTrace();
27         } // fim do catch
28
29         application.shutdown();
30      } // fim de main
31   } // fim da classe SharedBufferTest2
```

```
Operation                                Buffer      Occupied
---------                                --------    ---------

Producer writes 1                        1           true

Producer tries to write.
Buffer full. Producer waits.             1           true

Consumer reads 1                         1           false

Producer writes 2                        2           true

Producer tries to write.
Buffer full. Producer waits.             2           true

Consumer reads 2                         2           false

Producer writes 3                        3           true

Consumer reads 3                         3           false

Producer writes 4                        4           true

Consumer reads 4                         4           false

Consumer tries to read.
Buffer empty. Consumer waits.            4           false

Producer writes 5                        5           true

Consumer reads 5                         5           false

Consumer tries to read.
Buffer empty. Consumer waits.            5           false

Producer writes 6                        6           true

Consumer reads 6                         6           false

Producer writes 7                        7           true

Consumer reads 7                         7           false

Producer writes 8                        8           true

Consumer reads 8                         8           false

Producer writes 9                        9           true
```

Figura 23.12 SharedBufferTest2 configura um aplicativo produtor/consumidor que utiliza um buffer sincronizado. (Parte 2 de 3.)

```
Consumer reads 9                    9           false

Producer writes 10                  10          true

Producer done producing.
Terminating Producer.
Consumer reads 10                   10          false

Consumer read values totaling 55.
Terminating Consumer.
```

Figura 23.12 SharedBufferTest2 configura um aplicativo produtor/consumidor que utiliza um buffer sincronizado. (Parte 3 de 3.)

23.8 Relacionamento de produtor/consumidor: buffer circular

O programa na Seção 23.7 utiliza a sincronização de thread para garantir que duas threads manipulem corretamente os dados em um buffer compartilhado. Entretanto, o aplicativo não pode apresentar um ótimo desempenho. Se as duas threads operarem a diferentes velocidades, uma delas gastará mais tempo (ou a maior parte dele) na espera. Por exemplo, no programa na Seção 23.7, compartilhamos uma única variável de inteiro entre as duas threads. Se a thread produtora produzir valores mais rápido do que a consumidora pode consumir, a thread produtora esperará a consumidora, porque não há nenhuma outra posição na memória onde colocar o próximo valor. De maneira semelhante, se a consumidora consumir valores mais rapidamente do que a produtora os produz, a consumidora esperará até que a produtora coloque o próximo valor na posição compartilhada na memória. Mesmo quando temos threads que operam nas mesmas velocidades relativas, essas threads podem ficar ocasionalmente 'fora de sincronia' por um período de tempo, fazendo com que uma delas espere a outra. Não podemos fazer suposições a respeito das velocidades relativas de threads concorrentes — as interações que ocorrem com o sistema operacional, a rede, o usuário e outros componentes podem fazer com que as threads operem a diferentes velocidades. Quando isso acontece, as threads esperam. Quando as threads esperam excessivamente, os programas tornam-se menos eficientes, os programas interativos com usuários tornam-se menos responsivos e os aplicativos sofrem retardos mais longos.

Para minimizar a quantidade de tempo de espera por threads que compartilham recursos e operam nas mesmas velocidades médias, podemos implementar um **buffer circular**, que fornece espaço de buffer extra em que a produtora pode colocar valores e a partir do qual a consumidora pode recuperá-los. Vamos supor que o buffer é implementado como um array e que a produtora e a consumidora funcionam bem desde o início do array. Quando qualquer thread alcançar o fim do array, ela simplesmente retorna ao primeiro elemento do array para realizar sua próxima tarefa. Se a produtora produzir valores temporariamente mais rápido do que a consumidora pode consumi-los, a produtora pode escrever valores adicionais no espaço extra de buffer (se algum estiver disponível). Essa capacidade permite à produtora realizar sua tarefa, mesmo que a consumidora não esteja pronta para receber o valor atual sendo produzido. De maneira semelhante, se a consumidora consumir mais rápido do que a capacidade da produtora de produzir novos valores, a consumidora poderá ler valores adicionais (se houver quaisquer valores) do buffer. Isso permite que a consumidora se mantenha ocupada mesmo que a produtora não esteja pronta para produzir valores adicionais.

Observe que o buffer circular seria inadequado se a produtora e a consumidora operassem consistentemente em diferentes velocidades. Se a consumidora sempre executar mais rápido do que a produtora, então um buffer que contém uma única posição será suficiente — posições adicionais desperdiçariam memória. Se a produtora sempre executar mais rápido, somente um buffer com um número infinito de posições seria capaz de absorver a produção extra.

A chave para utilizar um buffer circular com uma produtora e uma consumidora que operam quase na mesma velocidade é fornecer ao buffer posições suficientes para tratar a produção 'extra' antecipada. Se, em um período de tempo, determinarmos que a produtora produz freqüentemente até mais três valores do que a consumidora pode consumir, podemos fornecer um buffer de pelo menos três células para tratar a produção extra. Não queremos que o buffer seja muito pequeno, porque isso faria com que as threads esperassem mais tempo. Por outro lado, o buffer não pode ser muito grande, porque isso desperdiçaria memória.

Dica de desempenho 23.4

Mesmo ao utilizar um buffer circular, é possível que uma thread produtora pudesse preencher o buffer, o que forçaria a thread produtora a esperar até que uma consumidora consumisse um valor para liberar um elemento no buffer. De maneira semelhante, se o buffer estiver vazio em qualquer dado momento, a thread consumidora deve esperar até que a produtora produza outro valor. A chave para utilizar um buffer circular é otimizar o tamanho do buffer para minimizar a quantidade de tempo de espera da thread.

O programa nas figuras 23.13– 23.14 mostra uma produtora e uma consumidora acessando um buffer circular (nesse caso, um array compartilhado de três células) com a sincronização. Nessa versão do relacionamento produtor/consumidor, a consumidora só consome um valor quando o array não estiver vazio e a produtora só produz um valor quando o array não estiver cheio. As instruções que criaram e iniciaram os objetos de thread no método `main` da classe `SharedBufferTest2` (Figura 23.12) agora aparecem na classe `CircularBufferTest` (Figura 23.14).

```
1  // Fig. 23.13: CircularBuffer.java
2  // SynchronizedBuffer sincroniza o acesso a um único inteiro compartilhado.
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5  import java.util.concurrent.locks.Condition;
6
7  public class CircularBuffer implements Buffer
8  {
9     // Bloqueio para controlar a sincronização com esse buffer
10    private Lock accessLock = new ReentrantLock();
11
12    // condições para controlar leitura e gravação
13    private Condition canWrite = accessLock.newCondition();
14    private Condition canRead = accessLock.newCondition();
15
16    private int[] buffer = { -1, -1, -1 };
17
18    private int occupiedBuffers = 0; // conta o número de buffers utilizados
19    private int writeIndex = 0; // índice para escrever o próximo valor
20    private int readIndex = 0; // índice para ler o próximo valor
21
22    // coloca o valor no buffer
23    public void set( int value )
24    {
25       accessLock.lock(); // bloqueia esse objeto
26
27       // envia informações de thread e de buffer para a saída, então espera
28       try
29       {
30          // enquanto não houver posições vazias, põe o thread no estado de espera
31          while ( occupiedBuffers == buffer.length )
32          {
33             System.out.printf( "All buffers full. Producer waits.\n" );
34             canWrite.await();// espera até um elemento buffer ser liberado
35          } // fim do while
36
37          buffer[ writeIndex ] = value; // configura novo valor de buffer
38
39          // atualiza índice de gravação circular
40          writeIndex = ( writeIndex + 1 ) % buffer.length;
41
42          occupiedBuffers++; // mais um elemento buffer está cheio
43          displayState( "Producer writes " + buffer[ writeIndex ] );
44          canRead.signal(); // sinaliza threads que estão esperando para ler o buffer
45       } // fim do try
46       catch ( InterruptedException exception )
47       {
48          exception.printStackTrace();
49       } // fim do catch
50       finally
51       {
52          accessLock.unlock(); // desbloqueia esse objeto
53       } // fim de finally
54    } // fim do método set
```

Figura 23.13 `CircularBuffer` sincroniza acesso a um buffer circular que contém três posições. (Parte 1 de 3.)

```
55
56    // retorna valor do buffer
57    public int get()
58    {
59       int readValue = 0; // inicializa valor lido a partir do buffer
60       accessLock.lock(); // bloqueia esse objeto
61
62       // espera até que o buffer tenha dados, então lê o valor
63       try
64       {
65          // enquanto os dados não são lidos, coloca thread em estado de espera
66          while ( occupiedBuffers == 0 )
67          {
68             System.out.printf( "All buffers empty. Consumer waits.\n" );
69             canRead.await(); // espera até que um elemento buffer seja preenchido
70          } // fim do while
71
72          readValue = buffer[ readIndex ]; // lê valor do buffer
73
74          // atualiza índice de leitura circular
75          readIndex = ( readIndex + 1 ) % buffer.length;
76
77          occupiedBuffers--; // mais um elemento buffer está vazio
78          displayState( "Consumer reads " + readValue );
79          canWrite.signal(); // sinaliza threads que estão esperando para gravar no buffer
80       } // fim do try
81       // se a thread na espera tiver sido interrompida, imprime o rastreamento de pilha
82       catch ( InterruptedException exception )
83       {
84          exception.printStackTrace();
85       } // fim do catch
86       finally
87       {
88          accessLock.unlock(); // desbloqueia esse objeto
89       } // fim de finally
90
91       return readValue;
92    } // fim do método get
93
94    // exibe a operação atual e o estado de buffer
95    public void displayState( String operation )
96    {
97       // gera saída de operação e número de buffers ocupados
98       System.out.printf( "%s%s%d)\n%s", operation,
99          " (buffers occupied: ", occupiedBuffers, "buffers:  " );
100
101      for ( int value : buffer )
102         System.out.printf( " %2d  ", value ); // gera a saída dos valores no buffer
103
104      System.out.print( "\n          " );
105      for ( int i = 0; i < buffer.length; i++ )
106         System.out.print( "---- " );
107
108      System.out.print( "\n          " );
109      for ( int i = 0; i < buffer.length; i++ )
```

Figura 23.13 `CircularBuffer` sincroniza acesso a um buffer circular que contém três posições. (Parte 2 de 3.)

```
110       {
111          if ( i == writeIndex && i == readIndex )
112             System.out.print( " WR" ); // índice de gravação e leitura
113          else if ( i == writeIndex )
114             System.out.print( " W  " ); // só grava índice
115          else if ( i == readIndex )
116             System.out.print( "  R " ); // só lê índice
117          else
118             System.out.print( "     " ); // nenhum dos índices
119       } // fim do for
120
121       System.out.println( "\n" );
122    } // fim do método displayState
123 } // fim da classe CircularBuffer
```

Figura 23.13 `CircularBuffer` sincroniza acesso a um buffer circular que contém três posições. (Parte 3 de 3.)

As alterações significativas no exemplo da Seção 23.7 ocorrem em `CircularBuffer` (Figura 23.13), que substitui `SynchronizedBuffer` (Figura 23.11). A linha 10 cria um novo objeto `ReentrantLock` e atribui sua referência à variável `Lock accessLock`. O `ReentrantLock` é criado sem a diretiva de imparcialidade porque temos somente duas threads nesse exemplo e apenas uma estará sempre esperando. As linhas 13–14 criam duas `Conditions` utilizando o método `Lock newCondition`. `Condition canWrite` contém uma fila de threads que esperam o buffer tornar-se cheio. Se o buffer estiver cheio, `Producer` chama o método `await` nessa `Condition` — quando `Consumer` liberar espaço em um buffer cheio, ela chama o método `signal` nessa `Condition`. `Condition canRead` contém uma fila de threads que esperam enquanto o buffer está vazio. Se o buffer estiver vazio, `Consumer` chama o método `await` nessa `Condition` — quando `Producer` gravar no buffer, ela chamará o método `signal` nessa `Condition`. O array `buffer` (linha 16) é um array de inteiros de três elementos que representam o buffer circular. A variável `occupiedBuffers` (linha 18) conta o número de elementos no `buffer` que são preenchidos com dados disponíveis para leitura. Quando `occupiedBuffers` for `0`, não haverá dados no buffer circular e `Consumer` deverá esperar — quando `occupiedBuffers` for `3` (o tamanho do buffer circular), o buffer circular estará cheio e `Producer` deverá esperar. A variável `writeIndex` (linha 19) indica a próxima posição em que um valor pode ser colocado por uma `Producer`. A variável `readIndex` (linha 20) indica a posição a partir da qual o próximo valor pode ser lido por um método `Consumer`.

`CircularBuffer set` (linhas 23–54) realiza as mesmas tarefas que ele realizou na Figura 23.11, com algumas modificações. O loop `while` nas linhas 31–35 determina se `Producer` precisa esperar (isto é, todos os buffers estão cheios). Se precisar, a linha 33 indica que `Producer` está esperando para realizar sua tarefa. Então a linha 34 invoca método `Condition await` para colocar a thread `Producer` no estado de *espera* na variável de condição `canWrite`. Quando a execução finalmente continuar na linha 37 depois do loop `while`, o valor gravado pela `Producer` será colocado no buffer circular na posição `writeIndex`. Então a linha 40 atualiza `writeIndex` para a próxima chamada para o método `CircularBuffer set`. Essa linha é a chave para a circularidade do buffer. Quando `writeIndex` é incrementado depois do final do buffer, essa linha o configura como `0`. A linha 42 incrementa `occupiedBuffers`, uma vez que agora há pelo menos um valor no buffer que `Consumer` pode ler. Em seguida, a linha 43 invoca o método `displayState` para atualizar a saída com o valor produzido, o número de buffers ocupados, o conteúdo dos buffers e os `writeIndex` e `readIndex` atuais. A linha 44 invoca o método `Condition signal` para indicar que uma thread `Consumer` está esperando na variável de condição `canRead` (se houver uma thread na espera) deve fazer uma transição para o estado *executável*. A linha 52 libera `accessLock` chamando o método `unlock` dentro de um bloco `finally`.

O método `get` (linhas 57–92) da classe `CircularBuffer` também realiza as mesmas tarefas que ele realizou na Figura 23.11, com algumas pequenas modificações. O loop `while` nas linhas 66–70 determina se a thread `Consumer` deve esperar (isto é, todos os buffers estão vazios). Se precisar, a linha 68 atualiza a saída para indicar que `Consumer` está esperando para realizar sua tarefa. Então a linha 69 invoca o método `Condition await` para colocar a thread atual no estado de *espera* na variável de condição `canRead`. Quando a execução por fim continuar na linha 72 depois de uma chamada `signal` de `Producer`, `readValue` recebe o valor na localização `readIndex` no buffer circular. Então a linha 75 atualiza `readIndex` para a próxima chamada para o método `CircularBuffer get`. Essa linha e a linha 40 criam o efeito circular do buffer. A linha 77 decrementa os `occupiedBuffers`, porque há pelo menos uma posição aberta no buffer em que a thread `Producer` pode colocar um valor. A linha 78 invoca o método `displayState` para atualizar a saída com o valor consumido, o número de buffers ocupados, o conteúdo dos buffers e o `writeIndex` e `readIndex` atuais. A linha 79 invoca o método `Condition signal` para fazer a transição da thread que espera gravar no objeto `CircularBuffer` no estado *executável*. A linha 88 libera `accessLock` dentro de um bloco `finally` para garantir que o bloqueio seja liberado. Então a linha 91 retorna o valor consumido ao método chamador.

O método `displayState` (linhas 95–122) gera saída do estado do aplicativo. As linhas 101–102 geram saída dos buffers atuais. A linha 102 utiliza o método `printf` com um especificador de formato `%2d` para imprimir o conteúdo de cada buffer com um espaço inicial, se ele tiver um único dígito. As linhas 109–119 geram saída dos `writeIndex` e `readIndex` atuais com as letras `W` e `R`, respectivamente.

A classe `CircularBufferTest` (Figura 23.14) contém o método `main` que carrega o aplicativo. A linha 11 cria o `ExecutorService` com duas threads, e a linha 14 cria um objeto `CircularBuffer` e atribui sua referência à variável `Buffer sharedLocation`. As linhas 18–19 executam `Producer` e `Consumer`. A linha 26 chama o método `shutdown` para encerrar o aplicativo assim que `Producer` e `Consumer` completarem suas tarefas.

```
1  // Fig. 23.14: CircularBufferTest.java
2  // Aplicativo mostra duas threads que manipulam um buffer circular.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class CircularBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // cria CircularBuffer para armazenar ints
14       Buffer sharedLocation = new CircularBuffer();
15
16       try // tenta iniciar a produtora e a consumidora
17       {
18          application.execute( new Producer( sharedLocation ) );
19          application.execute( new Consumer( sharedLocation ) );
20       } // fim do try
21       catch ( Exception exception )
22       {
23          exception.printStackTrace();
24       } // fim do catch
25
26       application.shutdown();
27    } // fim de main
28 } // fim da classe CircularBufferTest
```

```
Producer writes 1 (buffers occupied: 1)
buffers:    1   -1   -1
          ---- ---- ----
            R   W

Consumer reads 1 (buffers occupied: 0)
buffers:    1   -1   -1
          ---- ---- ----
               WR

All buffers empty. Consumer waits.
Producer writes 2 (buffers occupied: 1)
buffers:    1    2   -1
          ---- ---- ----
                R    W

Consumer reads 2 (buffers occupied: 0)
buffers:    1    2   -1
          ---- ---- ----
                    WR

Producer writes 3 (buffers occupied: 1)
buffers:    1    2    3
          ---- ---- ----
            W         R

Consumer reads 3 (buffers occupied: 0)
buffers:    1    2    3
          ---- ---- ----
           WR
```

Figura 23.14 `CircularBufferTest` configura um aplicativo produtor/consumidor e instancia as threads produtora e consumidora. (Parte 1 de 3.)

```
Producer writes 4 (buffers occupied: 1)
buffers:    4    2    3
         ----- ----- -----
            R    W

Producer writes 5 (buffers occupied: 2)
buffers:    4    5    3
         ----- ----- -----
            R         W

Consumer reads 4 (buffers occupied: 1)
buffers:    4    5    3
         ----- ----- -----
                 R    W

Producer writes 6 (buffers occupied: 2)
buffers:    4    5    6
         ----- ----- -----
            W    R

Producer writes 7 (buffers occupied: 3)
buffers:    7    5    6
         ----- ----- -----
                WR

Consumer reads 5 (buffers occupied: 2)
buffers:    7    5    6
          ----- ----- -----
                 W    R

Producer writes 8 (buffers occupied: 3)
buffers:    7    8    6
          ----- ----- -----
                      WR

Consumer reads 6 (buffers occupied: 2)
buffers:    7    8    6
         ----- ----- -----
            R         W


Consumer reads 7 (buffers occupied: 1)
buffers:    7    8    6
         ----- ----- -----
                 R    W

Producer writes 9 (buffers occupied: 2)
buffers:    7    8    9
         ----- ----- -----
            W    R

Consumer reads 8 (buffers occupied: 1)
buffers:    7    8    9
         ----- ----- -----
            W         R

Consumer reads 9 (buffers occupied: 0)
buffers:    7    8    9
         ----- ----- -----
          WR

Producer writes 10 (buffers occupied: 1)
buffers:   10    8    9
         ----- ----- -----
            R   W
```

Figura 23.14 `CircularBufferTest` configura um aplicativo produtor/consumidor e instancia as threads produtora e consumidora. (Parte 2 de 3.)

```
Producer done producing.
Terminating Producer.
Consumer reads 10 (buffers occupied: 0)
buffers:   10    8    9
          ---- ---- ----
               WR


Consumer read values totaling: 55.
Terminating Consumer.
```

Figura 23.14 `CircularBufferTest` configura um aplicativo produtor/consumidor e instancia as threads produtora e consumidora. (Parte 3 de 3.)

Toda vez que `Producer` grava um valor ou `Consumer` lê um valor, o programa gera saída da ação realizada (uma leitura ou gravação) junto com o conteúdo do buffer e a localização dos índices de gravação e de leitura. Nessa saída, `Producer` grava primeiro o valor 1. O buffer então contém o valor 1 na primeira posição e o valor -1 (o valor padrão) nas outras duas posições. O índice de gravação é atualizado para a segunda posição, enquanto o índice de leitura permanece na primeira posição. Em seguida, `Consumer` lê 1. O buffer contém os mesmos valores, mas o índice de leitura foi atualizado na segunda posição. `Consumer` então tenta ler novamente, mas o buffer está vazio e ela tem de esperar. Observe que durante essa execução foi necessário que uma thread esperasse apenas uma vez.

23.9 Relacionamento de produtor/consumidor: ArrayBlockingQueue

O J2SE 5.0 inclui uma classe de buffer circular completamente implementada chamada **`ArrayBlockingQueue`** no pacote `java.util.concurrent`, que implementa a interface **`BlockingQueue`**. A interface **`BlockingQueue`**, por sua vez, implementa a interface `Queue`, discutida no Capítulo 19, e declara os métodos **`put`** e **`take`**, os equivalentes de bloqueio dos métodos `Queue` `offer` e `poll`, respectivamente. Isso significa que o método `put` colocará um elemento no fim do `BlockingQueue`, esperando se a fila estiver cheia. O método `take` removerá um elemento da cabeça da `BlockingQueue`, esperando se a fila estiver vazia. A classe `ArrayBlockingQueue` implementa a interface `BlockingQueue` que utiliza um array. Isso faz com que a estrutura de dados tenha um tamanho fixo, o que significa que não expandirá para acomodar elementos extras. A classe `ArrayBlockingQueue` encapsula todas as funcionalidades de nossa classe de buffer circular (Figura 23.13).

O programa nas figuras 23.15–23.16 mostra uma `Producer` e uma `Consumer` acessando um buffer circular (nesse caso, uma `ArrayBlockingQueue`) com sincronização. A classe `BlockingBuffer` implementa a interface `Buffer` (Figura 23.15) e contém uma variável de instância `ArrayBlockingQueue` que armazena objetos `Integer` (linha 7). Escolhendo implementar `Buffer`, nosso aplicativo pode reutilizar as classes `Producer` (Figura 23.7) e `Consumer` (Figura 23.8).

```
1  // Fig. 23.15: BlockingBuffer.java
2  // Classe sincroniza acesso a um buffer de bloqueio.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer
6  {
7     private ArrayBlockingQueue<Integer> buffer;
8
9     public BlockingBuffer()
10    {
11       buffer = new ArrayBlockingQueue<Integer>( 3 );
12    } // fim do construtor BlockingBuffer
13
14    // coloca o valor no buffer
15    public void set( int value )
16    {
17       try
18       {
19          buffer.put( value ); // coloca o valor no buffer circular
20          System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
21             "Buffers occupied: ", buffer.size() );
22       } // fim do try
```

Figura 23.15 `BlockingBuffer` cria um buffer circular de bloqueio para utilizar a classe `ArrayBlockingQueue`. (Parte 1 de 2.)

```
23         catch ( Exception exception )
24         {
25            exception.printStackTrace();
26         } // fim do catch
27      } // fim do método set
28
29      // retorna valor do buffer
30      public int get()
31      {
32         int readValue = 0; // inicializa o valor lido a partir do buffer
33
34         try
35         {
36            readValue = buffer.take(); // remove o valor do buffer circular
37            System.out.printf( "%s %2d\t%s%d\n", "Consumer reads ",
38               readValue, "Buffers occupied: ", buffer.size() );
39         } // fim do try
40         catch ( Exception exception )
41         {
42            exception.printStackTrace();
43         } // fim do catch
44
45         return readValue;
46      } // fim do método get
47   } // fim da classe BlockingBuffer
```

Figura 23.15 `BlockingBuffer` cria um buffer circular de bloqueio para utilizar a classe `ArrayBlockingQueue`. (Parte 2 de 2.)

A linha 19 no método `set` (linhas 15–27) chama o método `put` em `ArrayBlockingQueue`. Essa chamada de método bloqueará até que haja espaço no `buffer` para colocar o `value`. O método `get` (linhas 30–46) da classe `BlockingBuffer` chama o método `take` (linha 36) na `ArrayBlockingQueue`. Novamente, essa chamada de método bloqueará até que haja um elemento no `buffer` a ser removido. Observe que nenhum desses métodos requer um objeto `Lock` ou `Condition`. `ArrayBlockingQueue` trata toda a sincronização para você. A quantidade de código nesse programa diminui significativamente a partir do buffer circular anterior (de 123 linhas para 47 linhas) e é mais facilmente entendido. Esse é um excelente exemplo de encapsulamento e reutilização de software.

A classe `BlockingBufferTest` (Figura 23.16) contém o método `main` que carrega o aplicativo. A linha 11 cria `ExecutorService` e a linha 14 cria um objeto `BlockingBuffer` e atribui sua referência à variável `Buffer sharedLocation`. As linhas 18–19 executam `Producer` e `Consumer Runnables`. A linha 26 chama o método `shutdown` para encerrar o aplicativo quando `Producer` e `Consumer` terminarem.

Em nossos exemplos de sincronização anteriores, as instruções de saída nos métodos `set` e `get` do `Buffer` que indicavam o que `Producer` estava gravando ou o que `Consumer` estava lendo eram sempre executadas enquanto o bloqueio `Buffer` estava preso pela thread que chamavam `set` ou `get`. Isso garantia a ordem em que a saída seria exibida. Se a `Consumer` tivesse o bloqueio, a `Producer` não poderia executar o método `set` — portanto, não seria possível para a `Producer` gerar saída fora do turno. O inverso também era verdadeiro. Na Figura 23.15, os métodos `set` e `get` não utilizam mais bloqueios — todo bloqueio é tratado pela `ArrayBlockingQueue`. Como essa classe é da API do Java, não podemos modificá-la para realizar a saída de seus métodos `put` e `take`. Por essas razões, é possível que as instruções de saída de `Producer` e `Consumer` nesse exemplo sejam impressas fora de ordem. Mesmo que `ArrayBlockingQueue` esteja sincronizando adequadamente o acesso aos dados, as instruções de saída não são mais sincronizadas.

```
1  // Fig. 23.16: BlockingBufferTest.java
2  // Aplicativo mostra duas threads que manipulam um buffer de bloqueio.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class BlockingBufferTest
7  {
8     public static void main( String[] args )
9     {
```

Figura 23.16 `BlockingBufferTest` configura um aplicativo produtor/consumidor utilizando um buffer circular e bloqueio. (Parte 1 de 2.)

```
10        // cria novo pool de threads com duas threads
11        ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13        // cria BlockingBuffer para armazenar ints
14        Buffer sharedLocation = new BlockingBuffer();
15
16        try // tenta iniciar a produtora e a consumidora
17        {
18           application.execute( new Producer( sharedLocation ) );
19           application.execute( new Consumer( sharedLocation ) );
20        } // fim do try
21        catch ( Exception exception )
22        {
23           exception.printStackTrace();
24        } // fim do catch
25
26        application.shutdown();
27     } // fim de main
28  } // fim da classe BlockingBufferTest
```

```
Producer writes  1      Buffers occupied: 1
Consumer reads   1      Buffers occupied: 0
Producer writes  2      Buffers occupied: 1
Consumer reads   2      Buffers occupied: 0
Producer writes  3      Buffers occupied: 1
Consumer reads   3      Buffers occupied: 0
Producer writes  4      Buffers occupied: 1
Consumer reads   4      Buffers occupied: 0
Producer writes  5      Buffers occupied: 1
Consumer reads   5      Buffers occupied: 0
Producer writes  6      Buffers occupied: 1
Consumer reads   6      Buffers occupied: 0
Producer writes  7      Buffers occupied: 1
Producer writes  8      Buffers occupied: 2
Consumer reads   7      Buffers occupied: 1
Producer writes  9      Buffers occupied: 2
Consumer reads   8      Buffers occupied: 1
Producer writes 10      Buffers occupied: 2

Producer done producing.
Terminating Producer.
Consumer reads   9      Buffers occupied: 1
Consumer reads  10      Buffers occupied: 0

Consumer read values totaling 55.
Terminating Consumer.
```

Figura 23.16 `BlockingBufferTest` configura um aplicativo produtor/consumidor utilizando um buffer circular e bloqueio. (Parte 2 de 2.)

23.10 Multithreading com GUI

Esse programa utiliza threads separadas para modificar o conteúdo exibido em uma GUI Swing. A natureza da programação de múltiplas threads impede que o programador saiba exatamente quando uma thread executará. Os componentes Swing não são **seguros para thread** — se múltiplas threads manipulam um componente GUI Swing, os resultados podem não ser corretos. Todas as interações com os componentes GUI Swing devem ser realizadas como parte da **thread de despacho de evento** (também conhecida como **thread de tratamento de evento**). A classe `SwingUtilities` (pacote `javax.swing`) fornece o método `static` `invokeLater` para ajudar nesse processo. O método `invokeLater` especifica instruções de processamento de GUIs para executar posteriormente como parte da thread de despacho de evento. O método `invokeLater` recebe como seu argumento um objeto que implementa a interface `Runnable`. O método coloca `Runnable` como um evento na fila da thread de despacho de eventos. Esses eventos são processados na ordem em que aparecem na fila. Como somente uma thread trata esses eventos, pode-se garantir que a GUI será atualizada apropriadamente.

Nosso próximo exemplo (Figura 23.17) mostra esse conceito. Quando esse programa chama `invokeLater`, a atualização de componente GUI será enfileirada para a execução na thread de despacho de evento. O método `run` de `Runnable` então será invocado como parte da thread de despacho de evento para a realizar a saída e assegurar que a atualização do componente GUI ocorrerá de uma maneira segura para thread. Esse exemplo também mostra como **suspender** uma thread (isto é, impedir temporariamente sua execução) e como **retomar** uma thread suspensa.

A classe `RunnableObject` (Figura 23.17) implementa o método `run` da interface `Runnable` (linhas 26–74). A linha 29 utiliza o método `static Thread` **`currentThread`** para determinar a thread em execução atualmente e o método `Thread` **`getName`** para retornar seu nome. Cada thread em execução tem um nome padrão que inclui o número da thread (ver a saída da Figura 23.18). As linhas 31–73 são um loop infinito. [*Nota*: Nos primeiros capítulos dissemos que os loops infinitos não são uma boa prática de programação porque o aplicativo não terminará. Nesse caso, o loop infinito está em uma thread separada da thread principal. Quando a janela de aplicativo for fechada nesse exemplo, todas as threads criadas pela thread principal também são fechadas, incluindo aquelas (como esta) que estão executando loops infinitos.] Em cada iteração do loop, a thread dorme (`sleeps`) por um intervalo aleatório de 0 a 1 segundo (linha 36).

Quando a thread acorda, a linha 38 adquire o `Lock` nesse aplicativo. As linhas 41–44 fazem loop enquanto a variável `boolean` `suspended` for `true`. A linha 43 chama o método `await` em `Condition` `suspend` para liberar o `Lock` temporariamente e colocar essa thread no estado de *espera*. Quando essa thread é sinalizada (`signaled`), ela readquire o `Lock`, volta ao estado *executável* e libera o `Lock` (linha 48). Quando `suspended` for `false`, a thread deve retomar a execução. Se `suspended` ainda for `true`, o loop executa novamente.

```
1  // Fig. 23.17: RunnableObject.java
2  // Runnable que grava um caractere aleatório em um JLabel
3  import java.util.Random;
4  import java.util.concurrent.locks.Condition;
5  import java.util.concurrent.locks.Lock;
6  import javax.swing.JLabel;
7  import javax.swing.SwingUtilities;
8  import java.awt.Color;
9
10 public class RunnableObject implements Runnable
11 {
12    private static Random generator = new Random(); // para letras aleatórias
13    private Lock lockObject; // bloqueio de aplicativo; passado para o construtor
14    private Condition suspend; // utilizado para suspender e retomar thread
15    private boolean suspended = false; // true se a thread for suspensa
16    private JLabel output; // JLabel para a saída
17
18    public RunnableObject( Lock theLock, JLabel label )
19    {
20       lockObject = theLock; // armazena o Lock para o aplicativo
21       suspend = lockObject.newCondition(); // cria nova Condition
22       output = label; // armazena JLabel para gerar saída de caractere
23    } // fim do construtor RunnableObject
24
25    // coloca os caracteres aleatórios na GUI
26    public void run()
27    {
28       // obtém nome de thread em execução
29       final String threadName = Thread.currentThread().getName();
30
31       while ( true ) // loop infinito; será terminado de fora
32       {
33          try
34          {
35             // dorme por até 1 segundo
36             Thread.sleep( generator.nextInt( 1000 ) );
37
38             lockObject.lock(); // obtém o bloqueio
```

Figura 23.17 `RunnableObject` gera saída de uma letra maiúscula aleatória em um `JLabel` de modo que o usuário pode suspender e retomar a execução. (Parte 1 de 3.)

```
            try
            {
               while ( suspended ) // faz loop até não ser suspenso
               {
                  suspend.await(); // suspende a execução da thread
               } // fim do while
            } // fim do try
            finally
            {
               lockObject.unlock(); // desbloqueia o bloqueio
            } // fim de finally
         } // fim do try
         // se a thread foi interrompida durante espera/enquanto dormia
         catch ( InterruptedException exception )
         {
            exception.printStackTrace(); // imprime o rastreamento de pilha
         } // fim do catch

         // exibe o caractere no JLabel correspondente
         SwingUtilities.invokeLater(
            new Runnable()
            {
               // seleciona o caractere aleatório e o exibe
               public void run()
               {
                  // seleciona a letra maiúscula aleatória
                  char displayChar =
                     ( char ) ( generator.nextInt( 26 ) + 65 );

                  // gera saída de caractere em JLabel
                  output.setText( threadName + ": " + displayChar );
               } // fim do método run
            } // fim da classe inner
         ); // fim da chamada para SwingUtilities.invokeLater
      } // fim do while
   } // fim do método run

   // altera o estado suspenso/em execução
   public void toggle()
   {
      suspended = !suspended; // alterna booleano que controla estado

      // muda cor de rótulo na suspensão/retomada
      output.setBackground( suspended ? Color.RED : Color.GREEN );

      lockObject.lock(); // obtém bloqueio
      try
      {
         if ( !suspended ) // se a thread foi retomada
         {
            suspend.signal(); // retoma a thread
         } // fim do if
      } // fim do try
      finally
```

Figura 23.17 `RunnableObject` gera saída de uma letra maiúscula aleatória em um `JLabel` de modo que o usuário pode suspender e retomar a execução. (Parte 2 de 3.)

```
93          {
94             lockObject.unlock(); // libera o bloqueio
95          } // fim de finally
96       } // fim do método toggle
97    } // fim da classe RunnableObject
```

Figura 23.17 RunnableObject gera saída de uma letra maiúscula aleatória em um JLabel, de modo que o usuário pode suspender e retomar a execução. (Parte 3 de 3.)

As linhas 58–72 chamam o método SwingUtilities invokeLater. As linhas 59–71 declaram uma classe interna anônima que implementa a interface Runnable, e as linhas 62–70 declaram o método run. A chamada de método para invokeLater coloca esse objeto Runnable em uma fila a ser executada pela thread de despacho de evento. As linhas 65–66 criam um caractere aleatório em letras maiúsculas. A linha 69 chama o método setText no JLabel output para exibir o nome de thread e o caractere aleatório no JLabel na janela de aplicativo.

Quando o usuário clica no JCheckBox à direita de um JLabel particular, o Thread correspondente deve ser suspenso (impedido temporariamente de executar) ou retomado (autorizado a continuar a execução). A suspensão e a retomada de uma thread podem ser implementadas com a sincronização de thread e os métodos await e signal de Condition. As linhas 77–96 declaram o método toggle, que alterará o estado de suspensão/retomada da thread atual. A linha 79 inverte o valor da variável boolean suspended. A linha 82 muda a cor de fundo do JLabel chamando o método setBackground. Se a thread for suspensa, a cor de fundo será Color.RED; se ela estiver executando, a cor de fundo será Color.GREEN. Como o método toggle é chamado do handler de evento na Figura 23.18, suas tarefas serão realizadas na thread de despacho de evento — portanto, não há nenhuma necessidade de utilizar invokeLater na linha 82. A linha 84 adquire o Lock desse aplicativo. A linha 87 então testa se a thread acabou de ser retomada. Se isso for verdadeiro, a linha 89 chama o método signal em Condition suspend. Essa chamada de método alertará uma thread que foi colocada no estado de *espera* pela chamada de método await na linha 43. A linha 94 libera o Lock nesse aplicativo dentro de um bloco finally.

Observe que a instrução if na linha 87 não tem um else associado. Se essa condição falhar, isso significa que a thread acabou de ser suspensa. Quando isso acontecer, uma thread que executa na linha 38 entrará no loop while e a linha 43 suspenderá a thread com uma chamada para o método await.

A classe RandomCharacters (Figura 23.18) exibe três JLabels e três JCheckBoxes. Uma thread de execução separada está associada com cada par JLabel e JCheckBox. Cada thread exibe letras do alfabeto aleatoriamente em seu objeto JLabel correspondente. A linha 33 cria um novo ExecutorService com o método newFixedThreadPool. As linhas 36–56 iteram três vezes. As linhas 38–41 criam e personalizam o JLabel. A linha 44 cria a JCheckBox, e a linha 47 adiciona um ActionListener a cada JCheckBox (isso será discutido mais adiante.) As linhas 51–52 criam um novo RunnableObject que implementa a interface Runnable. A linha 55 executa RunnableObject com uma das threads de execução criadas em runner na linha 33.

Se o usuário clicar na caixa de seleção Suspended ao lado de um JLabel particular, o programa invoca o método actionPerformed (linhas 65–74) para determinar a caixa de seleção que gerou o evento. As linhas 68–73 determinam a caixa de seleção que gerou o evento. A linha 71 verifica se a origem do evento é a JCheckBox no índice atual. Se for, a linha 72 chama o método toggle (linhas 75–92 da Figura 23.17) nesse RunnableObject.

```
1  // Fig. 23.18: RandomCharacters.java
2  // A classe RandomCharacters demonstra a interface Runnable
3  import java.awt.Color;
4  import java.awt.GridLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import java.util.concurrent.Executors;
8  import java.util.concurrent.ExecutorService;
9  import java.util.concurrent.locks.Condition;
10 import java.util.concurrent.locks.Lock;
11 import java.util.concurrent.locks.ReentrantLock;
12 import javax.swing.JCheckBox;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15
16 public class RandomCharacters extends JFrame implements ActionListener
17 {
18    private final static int SIZE = 3; // número de threads
```

Figura 23.18 RandomCharacters cria um JFrame com três RunnableObjects e três JCheckBoxes para permitir ao usuário suspender e retomar as threads. (Parte 1 de 3.)

```
19    private JCheckBox checkboxes[]; // array de JCheckBoxes
20    private Lock lockObject = new ReentrantLock( true ); // único bloqueio
21
22    // array de RunnableObjects para exibir caracteres aleatórios
23    private RunnableObject[] randomCharacters =
24       new RunnableObject[ SIZE ];
25
26    // configura GUI e arrays
27    public RandomCharacters()
28    {
29       checkboxes = new JCheckBox[ SIZE ]; // aloca espaço para array
30       setLayout( new GridLayout( SIZE, 2, 5, 5 ) ); // configura layout
31
32       // cria novo pool de threads com threads SIZE
33       ExecutorService runner = Executors.newFixedThreadPool( SIZE );
34
35       // loop itera SIZE vezes
36       for ( int count = 0; count < SIZE; count++ )
37       {
38          JLabel outputJLabel = new JLabel(); // cria JLabel
39          outputJLabel.setBackground( Color.GREEN ); // configura a cor
40          outputJLabel.setOpaque( true ); // configura JLabel para ser opaco
41          add( outputJLabel ); // adiciona JLabel a JFrame
42
43          // cria JCheckBox para controlar estado da suspensão/retomada
44          checkboxes[ count ] = new JCheckBox( "Suspended" );
45
46          // adiciona ouvinte que executa quando JCheckBox é clicado
47          checkboxes[ count ].addActionListener( this );
48          add( checkboxes[ count ] ); // adiciona JCheckBox a JFrame
49
50          // cria um novo RunnableObject
51          randomCharacters[ count ] =
52             new RunnableObject( lockObject, outputJLabel );
53
54          // executa RunnableObject
55          runner.execute( randomCharacters[ count ] );
56       } // fim do for
57
58       setSize( 275, 90 ); // configura o tamanho da janela
59       setVisible( true ); // mostra a janela
60
61       runner.shutdown(); // desliga quando as threads terminam
62    } // fim do construtor RandomCharacters
63
64    // trata os eventos JCheckBox
65    public void actionPerformed( ActionEvent event )
66    {
67       // faz loop em todas as JCheckBoxes no array
68       for ( int count = 0; count < checkboxes.length; count++ )
69       {
70          // verifica se essa JCheckBox foi a origem do evento
71          if ( event.getSource() == checkboxes[ count ] )
72             randomCharacters[ count ].toggle(); // alterna o estado
```

Figura 23.18 RandomCharacters cria um JFrame com três RunnableObjects e três JCheckBoxes para permitir ao usuário suspender e retomar as threads. (Parte 2 de 3.)

```
73          } // fim do for
74       } // fim do método actionPerformed
75
76       public static void main( String args[] )
77       {
78          // cria novo objeto RandomCharacters
79          RandomCharacters application = new RandomCharacters();
80
81          // configura aplicativo para encerrar quando a janela for fechada
82          application.setDefaultCloseOperation( EXIT_ON_CLOSE );
83       } // fim de main
84    } // fim da classe RandomCharacters
```

Figura 23.18 RandomCharacters cria um JFrame com três RunnableObjects e três JCheckBoxes para permitir ao usuário suspender e retomar as threads. (Parte 3 de 3.)

23.11 Outras classes e interfaces em java.util.concurrent

A interface Runnable fornece apenas as funcionalidades mais básicas para a programação de múltiplas threads. De fato, essa interface tem várias limitações. Suponha que um Runnable encontre um problema e tente lançar uma exceção verificada. O método run não é declarado para lançar nenhuma exceção, de modo que o problema deve ser tratado dentro do Runnable — a exceção não pode ser passada para a thread chamadora. Agora suponha que um Runnable esteja realizando um cálculo longo e o aplicativo queira recuperar o resultado desse cálculo. O método run não pode retornar um valor, então o aplicativo deve utilizar dados compartilhados para enviar o valor de volta para a thread chamadora. Isso também envolve o overhead de sincronizar acesso aos dados. Os desenvolvedores das novas APIs de concorrência em J2SE 5.0 reconheceram essas limitações e criaram uma nova interface para corrigi-las. A interface **Callable** (pacote java.util.concurrent) declara um único método chamado **call**. Essa interface é projetada para ser semelhante à interface Runnable — permitindo que uma ação seja realizada concorrentemente em uma thread separada —, mas o método call permite que a thread retorne um valor ou lance uma exceção verificada.

Um aplicativo que cria uma Callable provavelmente quer executar a Callable concorrentemente com outras Runnables e Callables. A interface ExecutorService fornece o método **submit**, que executará uma Callable passada como seu argumento. O método submit retorna um objeto do tipo **Future** (pacote java.util.concurrent), que é uma interface que representa a Callable em execução. A interface Future declara o método **get** para retornar o resultado da Callable e fornece outros métodos para gerenciar a execução de uma Callable.

23.12 Monitores e bloqueios de monitor

Outra maneira de realizar a sincronização é utilizar os **monitores** predefinidos do Java. Cada objeto tem um monitor, que permite que uma thread por vez execute dentro de uma **instrução synchronized** no objeto. Isso é realizado obtendo um bloqueio no objeto quando o programa entra na instrução synchronized. Essas instruções são declaradas utilizando a palavra-chave **synchronized** na forma

```
synchronized ( objeto )
{
   instruções
} // fim da instrução synchronized
```

onde *objeto* é o objeto cujo bloqueio de monitor será obtido. Se houver várias instruções synchronized tentando executar em um objeto ao mesmo tempo, apenas uma delas pode estar ativa no objeto por vez — todas as outras threads que tentarem entrar em uma instrução synchronized no mesmo objeto serão colocadas no estado *bloqueado*.

O estado *bloqueado* não está presente na Figura 23.1, mas faz transição para e a partir do estado *executável*. Quando uma thread *executável* tem de esperar para entrar em uma instrução synchronized, ela transita para o estado *bloqueado*. Quando a thread *bloqueada* entra na instrução synchronized, ela transita para o estado *executável*.

Quando uma instrução synchronized conclui sua execução, o bloqueio de monitor no objeto será liberado, e a thread *bloqueada* de prioridade mais alta que estiver tentando entrar em uma instrução synchronized prosseguirá. O Java também permite **métodos synchronized**. Um método synchronized é equivalente a um instrução synchronized para incluir o corpo inteiro de um método.

Observação de engenharia de software 23.3

O bloqueio que ocorre com a execução dos métodos `synchronized` poderia levar a um impasse se os bloqueios nunca fossem liberados. Quando ocorrem exceções, o mecanismo de exceção do Java coordena com o mecanismo de sincronização do Java para liberar bloqueios e evitar esses tipos de impasses.

Erro comum de programação 23.5

Ocorre um erro se um thread emite um `wait`, `notify` ou `notifyAll` sobre um objeto sem adquirir um bloqueio para ele. Isso causa uma `IllegalMonitorStateException`.

O aplicativo nas figuras 23.19 e 23.20 mostra uma produtora e uma consumidora que acessam um buffer compartilhado com sincronização. Nesse caso, a consumidora consome apenas depois que a produtora produzir um valor; a produtora, por sua vez, produz um novo valor somente depois que a consumidora consumir o valor produzido anteriormente. Nesse exemplo, reutilizamos a interface `Buffer` (Figura 23.6) e as classes `Producer` (Figura 23.7) e `Consumer` (Figura 23.8) do exemplo na Seção 23.6. O código que realiza a sincronização é colocado nos métodos `set` e `get` da classe `SynchronizedBuffer` (Figura 23.19), que implementa a interface `Buffer` (linha 4). Portanto, os métodos `run` de `Producer` e de `Consumer` simplesmente chamam os métodos `set` e `get` do objeto compartilhado, como no exemplo da Seção 23.6.

```
1  // Fig. 23.19: SynchronizedBuffer.java
2  // SynchronizedBuffer sincroniza acesso a um único inteiro compartilhado.
3
4  public class SynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // compartilhado pelas threads producer e consumer
7     private boolean occupied = false; // contagem de buffers ocupados
8
9     // coloca o valor no buffer
10    public synchronized void set( int value )
11    {
12       // enquanto não houver posições vazias, coloca a thread em estado de espera
13       while ( occupied )
14       {
15          // envia informações de thread e de buffer para a saída, então espera
16          try
17          {
18             System.out.println( "Producer tries to write." );
19             displayState( "Buffer full. Producer waits." );
20             wait();
21          } // fim do try
22          catch ( InterruptedException exception )
23          {
24             exception.printStackTrace();
25          } // fim do catch
26       } // fim do while
27
28       buffer = value; // configura novo valor de buffer
29
30       // indica que a produtora não pode armazenar outro valor
31       // até a consumidora recuperar valor atual de buffer
32       occupied = true;
33
34       displayState( "Producer writes " + buffer );
35
36       notify(); // instrui a thread em espera a entrar no estado executável
37    } // fim do método set; libera o bloqueio em SynchronizedBuffer
38
```

Figura 23.19 Sincronização do acesso aos dados compartilhados utilizando os métodos `Object` `wait` e `notify`. (Parte 1 de 2.)

```
39    // retorna valor do buffer
40    public synchronized int get()
41    {
42       // enquanto os dados não são lidos, coloca thread em estado de espera
43       while ( !occupied )
44       {
45          // envia informações de thread e de buffer para a saída, então espera
46          try
47          {
48             System.out.println( "Consumer tries to read." );
49             displayState( "Buffer empty. Consumer waits." );
50             wait();
51          } // fim do try
52          catch ( InterruptedException exception )
53          {
54             exception.printStackTrace();
55          } // fim do catch
56       } // fim do while
57
58       // indica que a produtora pode armazenar outro valor
59       // porque a consumidora acabou de recuperar o valor do buffer
60       occupied = false;
61
62       int readValue = buffer; // armazena valor em buffer
63       displayState( "Consumer reads " + readValue );
64
65       notify(); // instrui a thread em espera a entrar no estado executável
66
67       return readValue;
68    } // fim do método get; libera bloqueio em SynchronizedBuffer
69
70    // exibe a operação atual e o estado de buffer
71    public void displayState( String operation )
72    {
73       System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
74          occupied );
75    } // fim do método displayState
76 } // fim da classe SynchronizedBuffer
```

Figura 23.19 Sincronização do acesso aos dados compartilhados utilizando os métodos `Object` `wait` e `notify`. (Parte 2 de 2.)

A classe `SynchronizedBuffer` (Figura 23.19) contém dois campos — `buffer` (linha 6) e `occupied` (linha 7). O método `set` (linhas 10–37) e o método `get` (linhas 40–68) são declarados como métodos `synchronized` adicionando a palavra-chave `synchronized` entre o modificador de método e o tipo de retorno — portanto, somente uma thread pode chamar qualquer um desses métodos por vez em um objeto `SynchronizedBuffer` particular. O campo `occupied` é utilizado em expressões condicionais para determinar se é a vez da produtora ou da consumidora de realizar uma tarefa. Se `occupied` for `false`, `buffer` está vazio e a produtora pode chamar o método `set` para colocar um valor na variável `buffer`. Essa condição também significa que a consumidora não pode chamar o método `get` do `SynchronizedBuffer` para ler o valor de `buffer` porque ele está vazio. Se `occupied` for `true`, a consumidora pode chamar o método `get` do `SynchronizedBuffer` para ler um valor a partir da variável `buffer`, porque a variável contém novas informações. Essa condição também significa que a produtora não pode chamar o método `set` do `SynchronizedBuffer` para colocar um valor em `buffer`, porque o buffer está cheio atualmente.

Quando o método `run` da thread `Producer` invocar o método `synchronized` `set`, a thread tentará obter o bloqueio de monitor no objeto `SynchronizedBuffer`. Se o bloqueio de monitor estiver disponível, a thread `Producer` obtém o bloqueio. Então o loop `while` nas linhas 13–26 determina se `occupied` é `true`. Se for `true`, o buffer está cheio, então a linha 18 gera a saída de uma mensagem indicando que a thread `Producer` está tentando gravar um valor, e a linha 19 invoca o método `displayState` (linhas 71–75) para gerar a saída de outra mensagem indicando que o buffer está cheio e que a thread `Producer` está no estado de *espera*. A linha 20 invoca o método `wait` (herdado de `Object` por `SynchronizedBuffer`) para colocar a thread que chamou o método `set` (isto é, a thread `Producer`) no estado de *espera* pelo objeto `SynchronizedBuffer`. A chamada para `wait` faz com que a thread chamadora libere o

bloqueio no objeto `SynchronizedBuffer`. Isso é importante porque a thread não pode realizar atualmente sua tarefa e porque outras threads devem ter permissão de acessar o objeto nesse momento para permitir que a condição (`occupied`) mude. Agora outra thread pode tentar obter o bloqueio do objeto `SynchronizedBuffer` e invocar o método `set` ou `get` do objeto.

A thread produtora permanece no estado de *espera* até que seja notificada por outra thread de que pode prosseguir — ponto em que a thread produtora retorna ao estado *bloqueado* e tenta readquirir o bloqueio no objeto `SynchronizedBuffer`. Se o bloqueio estiver disponível, a thread produtora readquire o bloqueio e o método `set` continua a executar com a próxima instrução depois de `wait`. Como `wait` é chamado em um loop (linhas 13–26), a condição de continuação do loop é testada novamente para determinar se a thread pode prosseguir sua execução. Se não puder, `wait` é invocado novamente — caso contrário, o método `set` continua com a próxima instrução depois do loop.

A linha 28 no método `set` atribui `value` ao `buffer`. A linha 32 configura `occupied` como `true` para indicar que o `buffer` agora contém um valor (isto é, uma consumidora pode ler o valor e uma produtora ainda não pode colocar outro valor aí). A linha 34 invoca o método `displayState` para gerar a saída de uma linha para a janela console que indica que a produtora está gravando um novo valor no `buffer`. A linha 36 invoca o método `notify` (herdado de `Object`). Se houver qualquer thread em espera, a primeira entra no estado *bloqueado*, indicando que a thread agora pode tentar obter o bloqueio novamente. O método `notify` retorna imediatamente e o método `set` retorna ao seu chamador. Invocar o método `notify` funciona corretamente nesse programa porque somente uma thread chama o método `get` a qualquer hora (a `ConsumerThread`). Em programas que têm múltiplas threads esperando em uma condição, pode ser mais adequado utilizar o método `notifyAll` ou chamar o método `wait` com um tempo limite opcional. Quando o método `set` retorna, ele libera implicitamente o bloqueio na memória compartilhada.

Os métodos `get` e `set` são implementados de maneira semelhante. Quando o método `run` da thread `Consumer` invocar o método `synchronized get`, a thread tentará adquirir o bloqueio de monitor no objeto `SynchronizedBuffer`. Se o bloqueio estiver disponível, a thread `Consumer` o adquire. Então o loop `while` nas linhas 43–56 determina se `occupied` é `false`. Se for, significa que o buffer está vazio, então a linha 48 gera a saída de uma mensagem indicando que a thread `Consumer` está tentando ler um valor, e a linha 49 invoca o método `displayState` para gerar a saída de outra mensagem indicando que o buffer está vazio e que a thread `Consumer` está esperando. A linha 50 invoca o método `wait` para colocar a thread que chamou o método `get` (isto é, a thread `Consumer`) no estado de *espera* para o objeto `SynchronizedBuffer`. Novamente, a chamada para `wait` faz com que a thread chamadora libere o bloqueio no objeto `SynchronizedBuffer`, então outra thread pode tentar adquirir o bloqueio `SynchronizedBuffer` do objeto e invocar o método `set` ou `get` do objeto. Se o bloqueio no `SynchronizedBuffer` não estiver disponível (por exemplo, se a `ProducerThread` ainda não retornou do método `set`), a `ConsumerThread` é bloqueada até que o bloqueio se torne disponível.

A thread consumidora permanece no estado de *espera* até que a thread seja notificada por outra de que pode prosseguir — ponto em que a thread consumidora retorna ao estado *bloqueado* e tenta readquirir o bloqueio no objeto `SynchronizedBuffer`. Se o bloqueio estiver disponível, a thread consumidora readquire o bloqueio e o método `get` continua a executar com a próxima instrução depois de `wait`. Como `wait` é chamado em um loop (linhas 43–56), a condição de continuação do loop é testada novamente para determinar se a thread pode prosseguir sua execução. Se não puder, `wait` é invocado novamente — caso contrário, o método `get` continua com a próxima instrução depois do loop. A linha 60 configura `occupied` como `false` para indicar que agora o `buffer` está vazio (isto é, uma consumidora não pode ler o valor, mas uma produtora pode colocar outro valor no `buffer`), a linha 63 chama o método `displayState` para indicar que a consumidora está lendo e a linha 65 invoca o método `notify`. Se houver quaisquer threads no estado *bloqueado* para o bloqueio nesse objeto `SynchronizedBuffer`, uma delas entra no estado *executável*, indicando que a thread agora pode tentar readquirir o bloqueio e continuar a realizar sua tarefa. O método `notify` retorna imediatamente, então o método `get` retorna o valor de `buffer` ao seu chamador. A invocação do método `notify` funciona corretamente nesse programa porque somente uma thread chama o método `set` a qualquer hora (o `ProducerThread`). Os programas que têm múltiplas threads que esperam em uma condição devem invocar `notifyAll` para assegurar que múltiplas threads recebam as notificações adequadamente. Quando o método `get` retornar, o bloqueio no objeto `SynchronizedBuffer` será implicitamente liberado.

A classe `SharedBufferTest2` (Figura 23.20) é idêntica à classe `SharedBufferTest` (Figura 23.12). Analise as saídas na Figura 23.20. Observe que cada inteiro produzido é consumido exatamente uma vez — nenhum valor é perdido e nenhum valor é consumido mais de uma vez. A sincronização e a variável de condição asseguram que a produtora e a consumidora não podem realizar suas tarefas, a menos que seja a vez de alguma delas. A produtora deve ir primeiro, a consumidora deve esperar se a produtora não tiver produzido desde que a consumidora consumiu pela última vez, e a produtora deve esperar se a consumidora ainda não tiver consumido o valor mais recente que a produtora produziu. Execute esse programa várias vezes para confirmar que todo inteiro produzido é consumido exatamente uma vez. Na saída de exemplo, observe as linhas que indicam quando a produtora e a consumidora devem esperar para realizar suas respectivas tarefas.

```
1  // Fig. 23.20: SharedBufferTest2.java
2  // Aplicativo mostra duas threads que manipulam um buffer sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
```

Figura 23.20 `SharedBufferTest2` configura um aplicativo produtor/consumidor que utiliza um buffer sincronizado. (Parte 1 de 3.)

```
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // cria SynchronizedBuffer para armazenar ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t--------" );
18
19       try // tenta iniciar a produtora e a consumidora
20       {
21          application.execute( new Producer( sharedLocation ) );
22          application.execute( new Consumer( sharedLocation ) );
23       } // fim do try
24       catch ( Exception exception )
25       {
26          exception.printStackTrace();
27       } // fim do catch
28
29       application.shutdown();
30    } // fim de main
31 } // fim da classe SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------

Consumer tries to read.
Buffer empty. Consumer waits.           -1               false

Producer writes 1                        1               true

Consumer reads 1                         1               false

Consumer tries to read.
Buffer empty. Consumer waits.            1               false

Producer writes 2                        2               true

Consumer reads 2                         2               false

Producer writes 3                        3               true

Consumer reads 3                         3               false

Consumer tries to read.
Buffer empty. Consumer waits.            3               false

Producer writes 4                        4               true

Consumer reads 4                         4               false

Consumer tries to read.
Buffer empty. Consumer waits.            4               false

Producer writes 5                        5               true

Consumer reads 5                         5               false
```

Figura 23.20 SharedBufferTest2 configura um aplicativo produtor/consumidor que utiliza um buffer sincronizado. (Parte 2 de 3.)

```
Producer writes 6                    6          true

Consumer reads 6                     6          false

Consumer tries to read.
Buffer empty. Consumer waits.        6          false

Producer writes 7                    7          true

Consumer reads 7                     7          false

Consumer tries to read.
Buffer empty. Consumer waits.        7          false

Producer writes 8                    8          true

Consumer reads 8                     8          false

Producer writes 9                    9          true

Producer tries to write.
Buffer full. Producer waits.         9          true

Consumer reads 9                     9          false

Producer writes 10                   10         true


Producer done producing.
Terminating Producer.
Consumer reads 10                    10         false


Consumer read values totaling 55.
Terminating Consumer.
```

Figura 23.20 `SharedBufferTest2` configura um aplicativo produtor/consumidor que utiliza um buffer sincronizado. (Parte 3 de 3.)

23.13 Conclusão

Este capítulo apresentou a nova API de concorrência do Java e demonstrou a poderosa técnica de sincronização de thread. Multithreading é um tópico avançado que é o assunto de muitos livros. Utilizamos as técnicas de multithreading introduzidas aqui novamente no Capítulo 24, Redes, para ajudar a construir servidores de múltiplas threads que podem interagir com múltiplos clientes concorrentemente.

Resumo

- Os computadores realizam operações concorrentemente, como compilar programas, enviar arquivos para uma impressora e receber mensagens de correio eletrônico em uma rede.
- Em geral, as linguagens de programação fornecem um conjunto de instruções de controle que permite que os programadores realizem apenas uma ação por vez.
- Historicamente, a concorrência foi implementada como primitivas de sistemas operacionais disponíveis somente para programadores de sistemas experientes.
- O Java disponibiliza a concorrência para o programador de aplicativos. O programador especifica os aplicativos que contêm threads de execução — cada thread designando uma parte de um programa que pode executar concorrentemente com outras threads. Essa capacidade é chamada multithreading.
- Uma nova thread inicia seu ciclo de vida no estado *novo*. Ela permanece no estado *novo* até que o programa inicie a thread, o que coloca a thread no estado *executável*.
- Uma thread *executável* entra no estado *terminado* ao completar sua tarefa ou, do contrário, ela termina.
- Às vezes a thread transita para o estado de *espera* enquanto espera outra thread realizar uma tarefa. Uma vez nesse estado, a thread só volta ao estado *executável* quando outra thread sinalizar a thread de espera para retomar a execução.
- Uma thread *executável* pode entrar no estado de *espera sincronizada* por um intervalo especificado de tempo. Uma thread nesse estado transita novamente para o estado *executável* quando esse intervalo de tempo expira. Uma thread pode transitar para o estado de *espera sincronizada* se

fornecer um intervalo de espera opcional quando ela estiver esperando outra thread realizar uma tarefa. Essa thread retornará ao estado *executável* quando ela for sinalizada por outra thread ou quando o intervalo sincronizado expirar — o que ocorrer primeiro. Outra maneira de colocar uma thread no estado de *espera sincronizada* é colocá-la para dormir.

- No nível de sistema operacional, o estado *executável* realmente inclui dois estados separados. Quando uma thread entra pela primeira vez no estado *executável* a partir do estado *novo*, ela está no estado *pronto*. Uma thread *pronta* entra no estado *de execução* (isto é, começa a executar) quando o sistema operacional a atribui a um processador. Quando o quantum da thread expirar, a thread retornará ao estado *pronto* e o sistema operacional atribuirá outra thread ao processador.
- Toda thread do Java tem uma prioridade no intervalo entre `MIN_PRIORITY` (1) e `MAX_PRIORITY` (10). Por padrão, toda thread recebe a prioridade `NORM_PRIORITY` (5).
- A maioria das plataformas do Java suporta o fracionamento de tempo. Sem o fracionamento de tempo, toda thread em um conjunto de threads de igual prioridade executa até sua conclusão, antes que outras threads de igual prioridade obtenham uma chance de executar. Com o fracionamento de tempo, toda thread recebe uma breve rajada de tempo de processador, ou quantum, durante o qual a thread pode executar. Quando o quantum expirar — mesmo se a thread não tiver terminado a execução —, o processador é tirado dessa thread e recebe a próxima thread de igual prioridade — se alguma estiver disponível.
- O scheduler de thread determina a thread que executa em seguida. Uma implementação simples manterá a thread de prioridade mais alta que está executando o tempo todo. Se houver mais de uma thread de prioridade mais alta, o scheduler assegura que todas essas threads sejam executadas por um quantum no modo rodízio.
- Multithreading no Java é realizado implementando a interface `Runnable`, que declara um único método chamado `run`.
- `Runnable`s são executadas utilizando uma classe que implementa a interface `Executor`, que declara um único método chamado `execute`.
- A interface `ExecutorService` é uma subinterface de `Executor` que declara vários métodos para gerenciar o ciclo de vida do `Executor`. Um objeto que implementa a interface `ExecutorService` pode ser criado com os métodos `static` declarados na classe método `Executors`.
- `ExecutorService` `shutdown` termina toda thread em uma interface `ExecutorService` logo que terminar de executar sua `Runnable`.
- Quando múltiplas threads compartilharem um objeto, resultados indeterminados podem ocorrer, a menos que o objeto compartilhado seja sincronizado adequadamente. A exclusão mútua permite ao programador realizar essa sincronização de thread.
- Uma vez que um `Lock` foi obtido por uma thread (chamando o método `lock`), o objeto `Lock` não permitirá que outra thread obtenha o bloqueio até que a primeira libere o `Lock` (chamando o método `unlock`). Quando uma thread chamar o método `unlock`, o bloqueio no objeto será liberado e a thread na espera da prioridade mais alta que tentar bloquear o objeto prosseguirá.
- Uma vez que uma thread obtém o bloqueio em um objeto, se ela determina que não pode continuar com sua tarefa até que alguma condição seja satisfeita, a thread pode esperar em uma variável de condição, dispensando-a assim da disputa pelo processador e liberando o bloqueio no objeto. As variáveis de condição são criadas chamando o método `Lock` `newCondition`, que retorna um objeto `Condition`.
- Uma thread pode chamar o método `await` em um objeto `Condition` para liberar o `Lock` associado e colocar essa thread no estado de *espera* enquanto outras threads tentam obter o `Lock`. Quando outra thread satisfizer a condição em que a primeira thread está *esperando*, ela pode chamar o método `Condition` `signal` para permitir que a thread em espera transite para o estado *executável* novamente. Se uma thread chamar o método `Condition` `signalAll`, então todas as threads que esperam o bloqueio tornam-se elegíveis para readquirir o bloqueio. Somente uma dessas threads pode obter o bloqueio no objeto por vez — outras que tentarem adquirir o mesmo bloqueio terão de esperar até que o bloqueio torne-se novamente disponível.
- Em um relacionamento produtor/consumidor, a parte produtora de um aplicativo gera dados e os armazena em um objeto compartilhado, e a parte consumidora de um aplicativo lê dados do objeto compartilhado. Em um relacionamento produtor/consumidor de múltiplas threads, uma thread produtora gera dados e os coloca em um objeto compartilhado chamado buffer. Uma thread consumidora lê dados do buffer.
- Para minimizar a quantidade de tempo de espera por threads que compartilham recursos e operam nas mesmas velocidades médias, utilize um buffer circular que fornece espaço extra de buffer, em que a produtora pode colocar valores e de que a consumidora pode recuperar esses valores.
- A interface `BlockingQueue` declara os métodos `put` e `take`, que são os equivalentes de bloqueio de métodos `Queue` `offer` e `remove`, respectivamente. Isso significa que o método `put` colocará um elemento no fim de `BlockingQueue`, esperando se a fila estiver cheia. O método `take` removerá um elemento da cabeça de `BlockingQueue`, esperando se a fila estiver vazia. A classe `ArrayBlockingQueue` implementa a interface `BlockingQueue` que utiliza um array. Isso faz com que a estrutura de dados tenha um tamanho fixo, o que significa que não expandirá para acomodar elementos extras.
- Os componentes Swing não são seguros para thread — se múltiplas threads manipulam um componente GUI Swing, os resultados podem não estar corretos. Todas as interações com componentes GUI Swing devem ser realizadas na thread de despacho de evento. A classe `SwingUtilities` fornece o método `static` `invokeLater` para ajudar nesse processo. O método `invokeLater` recebe como seu argumento um objeto que implementa a interface `Runnable` e realiza as atualizações de GUIs.
- A interface `Callable` declara um único método chamado `call`, que permite ao usuário retornar um valor ou lançar uma exceção verificada. A interface `ExecutorService` fornece o método `submit` que executará uma `Callable` passada como seu argumento. O método `submit` retorna um objeto do tipo `Future` que é uma interface que representa o `Callable` em execução. A interface `Future` declara o método `get` para retornar o resultado do `Callable` e fornece outros métodos para gerenciar a execução de `Callable`.
- Um monitor do objeto permite que uma thread por vez execute dentro de uma instrução `synchronized` nesse objeto.
- Quando uma thread *executável* tem de esperar para entrar em uma instrução `synchronized`, ela transita para o estado *bloqueado*. Quando a thread *bloqueada* entra na instrução `synchronized`, ela transita para o estado *executável*.
- O Java também permite métodos `synchronized` que são equivalentes a uma instrução `synchronized` que inclui o corpo inteiro do método.
- Uma vez que uma thread obtém o bloqueio de monitor em um objeto, se a thread determinar que não pode continuar com sua tarefa nesse objeto até que alguma condição seja satisfeita, ela pode chamar o método `Object` `wait`, liberando o bloqueio monitor no objeto.

- Quando uma thread que está executando uma instrução `synchronized` completar ou satisfizer a condição em que outra thread pode estar esperando, ela pode chamar o método `Object` `notify` para permitir que uma thread em espera transite para o estado *bloqueado* novamente.
- Se uma thread chamar `notifyAll`, então todas as threads que estão esperando o bloqueio de monitor tornam-se elegíveis para readquirir o bloqueio (isto é, todas transitam para o estado *bloqueado*).

Terminologia

adiamento indefinido
agendamento de rodízio
agendamento de thread
agendamento preemptivo
`ArrayBlockingQueue`, classe
`await`, método da interface `Condition`
`BlockingQueue`, interface
buffer
buffer circular
`call`, método da interface `Callable`
`Callable`, interface
coleta de lixo
concorrência
`Condition`, interface
consumidor
despachar uma thread
diretiva de imparcialidade de um bloqueio
estado *bloqueado*
estado de *espera*
estado de *espera sincronizada*
estado de *pronto*
estado de thread
estado *em execução*
estado *executável*
estado *novo*
estado *terminado*
exclusão mútua
`execute`, método da interface `Executor`
`Executor`, interface
`Executors`, classe
`ExecutorService`, interface
fração de tempo
`Future`, interface
`get`, método da interface `Future`
`IllegalMonitorStateException`, classe
impasse
inanição
`interrupt`, método da classe `Thread`
`InterruptedException`, classe
intervalo de adormecimento
`invokeLater`, método da classe `SwingUtilities`
`Lock`, interface
`lock`, método da interface `Lock`
método `put` da interface `BlockingQueue`
método `synchronized`
monitor
multithreading
`newCachedThreadPool`, método da classe `Executors`
`newCondition`, método da interface `Lock`
`newFixedThreadPool`, método da classe `Executors`
`notify`, método da classe `Object`
`notifyAll`, método da classe `Object`
obter bloqueio
operações paralelas
palavra-chave `synchronized`
pool de threads
prioridade de uma thread
produtor
produtor/consumidor, relacionamento
programação concorrente
quantum
`ReentrantLock`, classe
retomar uma thread
`run`, método da interface `Runnable`
`Runnable`, interface
scheduler de thread
`shutdown`, método da classe `ExecutorService`
`signal`, método da classe `Condition`
`signalAll`, método da classe `Condition`
sincronização
sincronização de threads
`sleep`, método da classe `Thread`
`submit`, método da classe `ExecutorService`
suspender uma thread
`SwingUtilities`, classe
`synchronized`, instrução
`take`, método da interface `BlockingQueue`
thread
thread adormecida
thread coletora de lixo
thread consumidora
thread de despacho de evento
thread principal
thread produtora
`Thread`, classe
`unlock`, método da interface `Lock`
variável de condição
vazamento de memória
`wait`, método da classe `Object`

Exercícios de revisão

23.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) C e C++ são linguagens de _____, enquanto o Java é uma linguagem de _____.
b) Uma thread entra no estado *terminado* quando _____.
c) Para pausar por um número designado de milissegundos e retomar a execução, uma thread deve chamar o método _____.
d) O método _____ da classe `Condition` move uma única thread no estado de *espera* de um objeto para o estado *executável*.
e) O método _____ da classe `Condition` move toda thread no estado de *espera* de um objeto para o estado *executável*.
f) Uma thread _____ entra no estado _____ quando ela completa sua tarefa ou, de outro modo, termina.
g) Uma thread *executável* pode entrar no estado _____ por um intervalo especificado de tempo.
h) No nível do sistema operacional, o estado *executável* realmente inclui dois estados separados, _____ e _____.
i) `Runnable`s são executadas utilizando uma classe que implementa a interface _____.
j) O método `ExecutorService` _____ termina cada thread em uma interface `ExecutorService` logo que terminar de executar sua `Runnable` atual, se houver alguma.
k) Uma thread pode chamar o método _____ em um objeto `Condition` para liberar o `Lock` associado e colocar essa thread no estado _____.
l) Em um relacionamento _____, a parte _____ de um aplicativo gera dados e os armazena em um objeto compartilhado, e a parte _____ de um aplicativo lê os dados do objeto compartilhado.
m) Para minimizar a quantidade de tempo de espera de threads que compartilham recursos e operam nas mesmas velocidades médias, utilize um(a) _____.
n) A classe _____ implementa a interface `BlockingQueue` que utiliza um array.
o) A palavra-chave _____ indica que somente uma thread por vez deve executar em um objeto.

23.2 Determine se cada uma das sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
a) Uma thread não é *executável* se estiver morta.
b) Em Java, uma thread *executável* de prioridade mais alta deve fazer preempção da thread de prioridade mais baixa.
c) Alguns sistemas operacionais utilizam o fracionamento de tempo com threads. Portanto, eles podem permitir às threads fazer preempção de threads de mesma prioridade.
d) Quando o quantum da thread expirar, a thread retorna ao estado de *execução* quando o sistema operacional a atribui a um processador.
e) Sem o fracionamento de tempo, toda thread em um conjunto de threads de igual prioridade executa até sua conclusão, antes que outras threads de igual prioridade tenham uma chance de executar.

Respostas dos exercícios de revisão

23.1 a) uma única thread, múltiplas threads. b) seu método `run` é encerrado. c) `Thread.sleep`. d) `signal`. e) `signalAll`. f) *executável*, *terminado*. g) *de espera sincronizada*. h) *pronto*, *em execução*. i) `Executor`. j) `shutdown`. k) `await`, *de espera*. l) produtor/consumidor, produtora, consumidora. m) buffer circular. n) `ArrayBlockingQueue`. o) `synchronized`.

23.2 a) Verdadeira. b) Verdadeira. c) Falsa. O fracionamento de tempo permita a uma thread executar até que sua fração de tempo (ou quantum) expire. Então outras threads de igual prioridade podem executar. d) Falsa. Quando o quantum de uma thread expira, a thread retorna ao estado *pronto* e o sistema operacional atribui outra thread ao processador. e) Verdadeira.

Exercícios

23.3 Determine se cada uma das sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
a) O método `sleep` não consome tempo de processador enquanto uma thread dorme.
b) Utilizar um `Lock` garante que o impasse não pode ocorrer.
c) Uma vez que um `Lock` foi obtido por uma thread, o objeto `Lock` não permitirá que outra thread obtenha o bloqueio até que a primeira o libere.
d) Os componentes Swing são seguros para thread.

23.4 Defina cada um dos seguintes termos.
a) thread
b) multithreading
c) estado *executável*
d) estado de *espera sincronizada*
e) agendamento preemptivo
f) interface `Runnable`
g) método `signal` da classe `Condition`
h) relacionamento de produtor/consumidor
i) quantum

23.5 Discuta cada um dos seguintes termos no contexto de mecanismos de thread do Java:
a) `Lock`
b) produtor
c) consumidor
d) `await`
e) `signal`
f) `Condition`

23.6 Dois problemas que podem ocorrer em sistemas como o Java, que permitem que threads esperem, são os impasses — em que uma ou mais threads esperarão eternamente por um evento que não pode ocorrer — e o adiamento indefinido — em que uma ou mais threads serão retardadas por algum tempo imprevisivelmente longo. Dê um exemplo de como cada um desses problemas podem ocorrer em um programa Java multithreaded.

23.7 Discuta a diferença entre o método `Condition` `await` sem argumentos e o método `Condition` `await` com um argumento de intervalo de tempo. Em particular, que estados fazem as threads entrar e como essas threads podem retornar ao estado executável?

23.8 Nomeie três threads que são automaticamente criadas pela Java Virtual Machine e discuta a finalidade de cada uma.

23.9 Elabore um programa que faça uma bola azul rebater em um `JPanel`. A bola deve começar a se mover com um evento `mousePressed`. Quando a bola atingir a borda do `JPanel`, ela deve rebater fora da borda e continuar na direção oposta. A bola deve ser atualizada com uma interface `Runnable`.

23.10 Modifique o programa no Exercício 23.9 para adicionar uma nova bola toda vez que o usuário clicar no mouse. Ofereça um mínimo de 20 bolas. Escolha a cor para cada nova bola aleatoriamente.

23.11 Modifique o programa no Exercício 23.10 para adicionar sombras. À medida que uma bola se move, desenhe uma oval sólida preta na parte inferior do `JPanel`. Você pode considerar a adicionar um efeito 3-D aumentando ou diminuindo o tamanho de cada bola quando ela atingir a borda do `JPanel`.

23.12 Modifique o programa dos exercícios 23.10 ou 23.11 para rebater as bolas quando elas colidirem. Deve ocorrer uma colisão entre duas bolas quando a distância entre os centros dessas bolas for menor que a soma de seus raios.