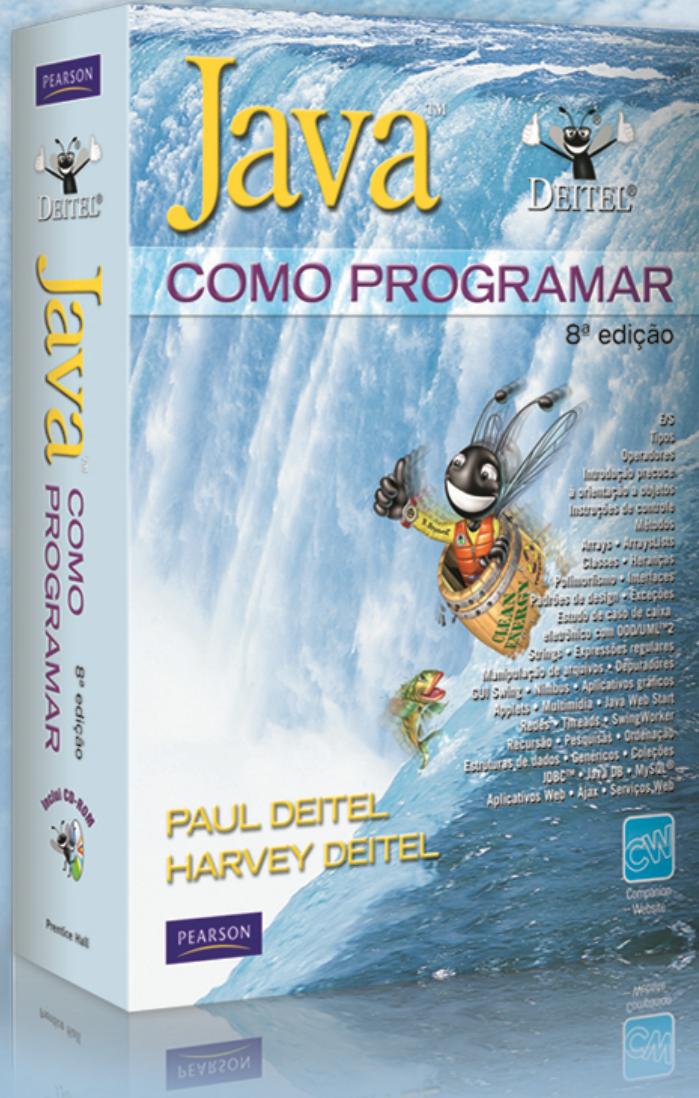


Capítulo 26

Multithreading

Java Como Programar, 8/E





COMO PROGRAMAR

8^a edição

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.
- As prioridades e agendamento de threads.
- A criar e executar `Runnables`.
- A sincronização de threads.
- O que são relacionamentos produtor/consumidor e como são implementados com multithreading.
- A permitir que múltiplas threads atualizem componentes GUI Swing de maneira segura para threads.
- Sobre as interfaces `Callable` e `Future`, que podem ser utilizadas com threading para executar tarefas que retornam resultados.



COMO PROGRAMAR

8^a edição

- 26.1** Introdução
- 26.2** Estados de thread: ciclo de vida de uma thread
- 26.3** Prioridades de thread e agendamento de thread
- 26.4** Criando e executando threads
 - 26.4.1 Runnables e a classe Thread
 - 26.4.2 Gerenciamento de threads com o framework Executor
- 26.5** Sincronização de thread
 - 26.5.1 Compartilhamento de dados não sincronizados
 - 26.5.2 Compartilhamento de dados sincronizados — tornando operações atômicas
- 26.6** Relacionamento entre produtor e consumidor sem sincronização
- 26.7** Relacionamento entre produtor e consumidor: ArrayBlockingQueue
- 26.8** Relacionamento entre produtor e consumidor com sincronização
- 26.9** Relacionamento entre produtor e consumidor: buffers limitados
- 26.10** Relacionamento entre produtor e consumidor: as interfaces Lock e Condition



COMO PROGRAMAR

8^a edição

26.11 Multithreading com GUI

26.11.1 Realizando cálculos em uma thread worker

26.11.2 Processando resultados intermediários com SwingWorker

26.12 Interfaces Callable e Future

26.13 Conclusão



COMO PROGRAMAR

8^a edição

26.1 Introdução

- ▶ Os sistemas operacionais em computadores de um único processador criam a ilusão da execução concorrente alternando rapidamente entre atividades, mas nesses computadores apenas uma instrução pode executar de cada vez.
- ▶ O Java disponibiliza a concorrência por meio da linguagem e das APIs.
- ▶ Você especifica que aplicativo contém **threads de execução** separados.
- ▶ Cada thread tem a sua própria pilha de chamadas de método e seu próprio contador de programas.
- ▶ Permite a execução simultânea com outras threads enquanto compartilha recursos no nível do aplicativo como a memória.
- ▶ Essa capacidade é chamada **multithreading**.



COMO PROGRAMAR

8^a edição



Dica de desempenho 26.1

Um problema com aplicativos de uma única thread que pode levar a uma fraca responsividade é que as atividades longas e demoradas devem ser concluídas antes de as outras poderem iniciar. Em um aplicativo com múltiplas threads, as threads podem ser distribuídas por múltiplos processadores (se disponíveis) de modo que múltiplas tarefas sejam mesmo executadas concorrentemente e o aplicativo possa operar de modo mais eficiente. O multithreading também pode aumentar o desempenho em sistemas de um único processador que simulam a concorrência — quando uma thread não puder avançar (porque, por exemplo, está esperando o resultado de uma operação E/S), outra pode utilizar o processador.



COMO PROGRAMAR

8^a edição

- ▶ É difícil e propenso a erros programar aplicativos concorrentes.
- ▶ Diretrizes:
 - Utilize classes existentes da Java API que gerenciam a sincronização para você.
 - Se precisar de capacidades ainda mais complexas, utilize as interfaces **Lock** e **Condition**.
 - As interfaces **Lock** e **Condition** só devem ser utilizadas por programadores avançados que conhecem as armadilhas comuns da programação concorrente.



COMO PROGRAMAR

8^a edição

26.2 Estados de thread: ciclo de vida de uma thread

- ▶ A qualquer dado momento, diz-se que uma thread está em um de vários **estados de thread** — ilustrados no diagrama de estado de UML na Figura 26.1.
- ▶ Uma nova thread inicia seu ciclo de vida no **estado novo**.
- ▶ Permanece nesse estado até ser iniciada, o que a coloca no estado **executável** — **considerado aquele que está executando sua tarefa**.
- ▶ Às vezes uma thread *executável* pode transitar para o estado de *espera* enquanto espera outra thread realizar uma tarefa.
- ▶ Uma thread de espera transita de volta para o estado executável apenas quando outra thread a notifica para continuar executando.



COMO PROGRAMAR

8^a edição

- ▶ Uma thread *executável* pode entrar no estado de *espera sincronizada* por um intervalo especificado de tempo.
- ▶ Ela transita para o estado *executável* quando esse intervalo de tempo expira ou quando o evento pelo qual ela está esperando ocorre.
- ▶ Não permite utilizar um processador, mesmo se houver um disponível.
- ▶ Uma **thread adormecida** permanece no estado de *espera sincronizada* por um período de tempo designado (chamado **intervalo de adormecimento**), depois do qual ela retorna ao estado *executável*.
- ▶ Uma thread *executável* transita para o estado **bloqueado** quando tenta realizar uma tarefa que não pode ser completada imediatamente e deve esperar temporariamente até que essa tarefa seja concluída.
- ▶ Uma thread *executável* entra no estado **terminado** quando completa sua tarefa ou, caso contrário, termina (talvez por causa de um erro).

COMO PROGRAMAR

8ª edição

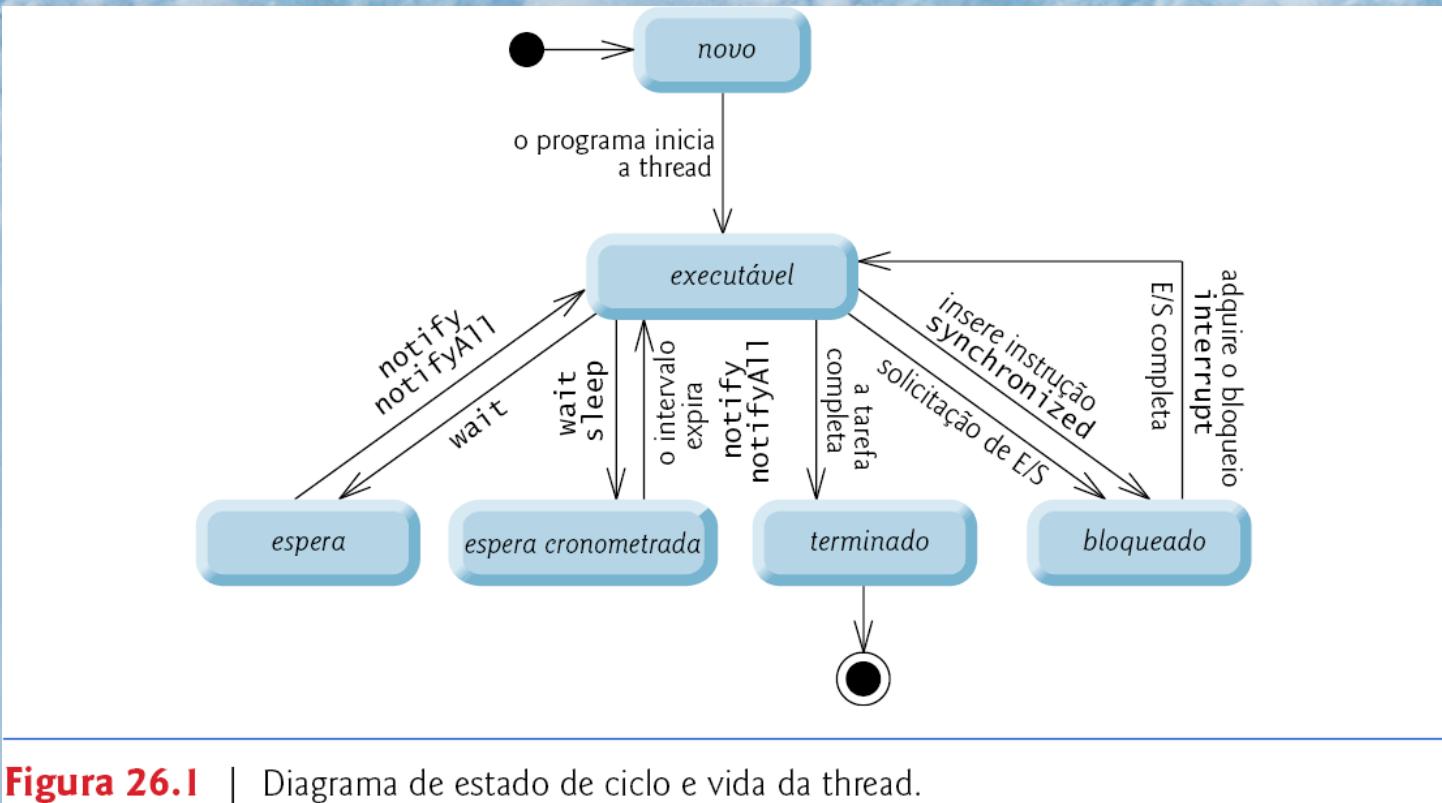


Figura 26.1 | Diagrama de estado de ciclo e vida da thread.



COMO PROGRAMAR

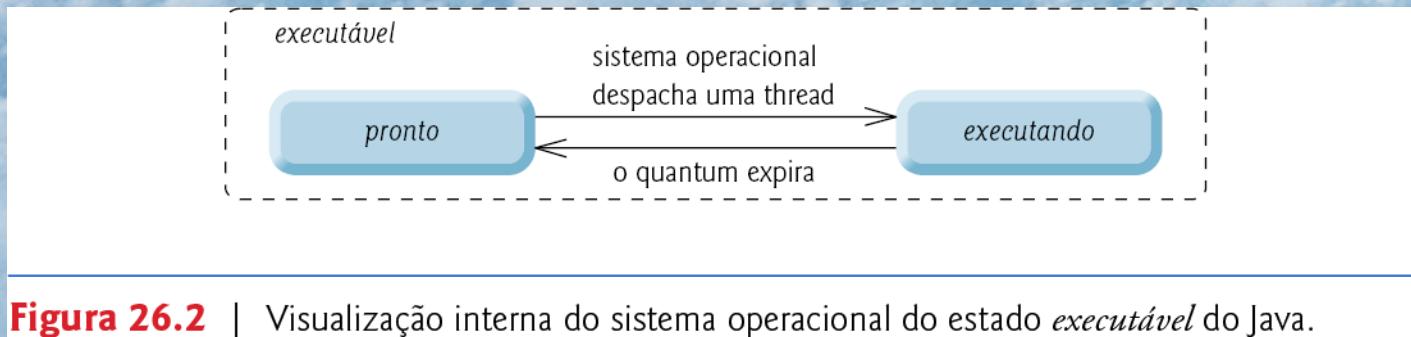
8^a edição

- ▶ No nível do sistema operacional, o estado *executável* do Java geralmente inclui dois estados separados (Figura 26.2).
 - ▶ 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 — também conhecido como *despachar a thread*.
- ▶ Em geral, cada thread recebe um **quantum** ou **fração de tempo** com o qual realizar sua tarefa.
- ▶ O processo que um sistema operacional utiliza para decidir qual thread despachar é conhecido como **agendamento de thread**.



COMO PROGRAMAR

8^a edição





COMO PROGRAMAR

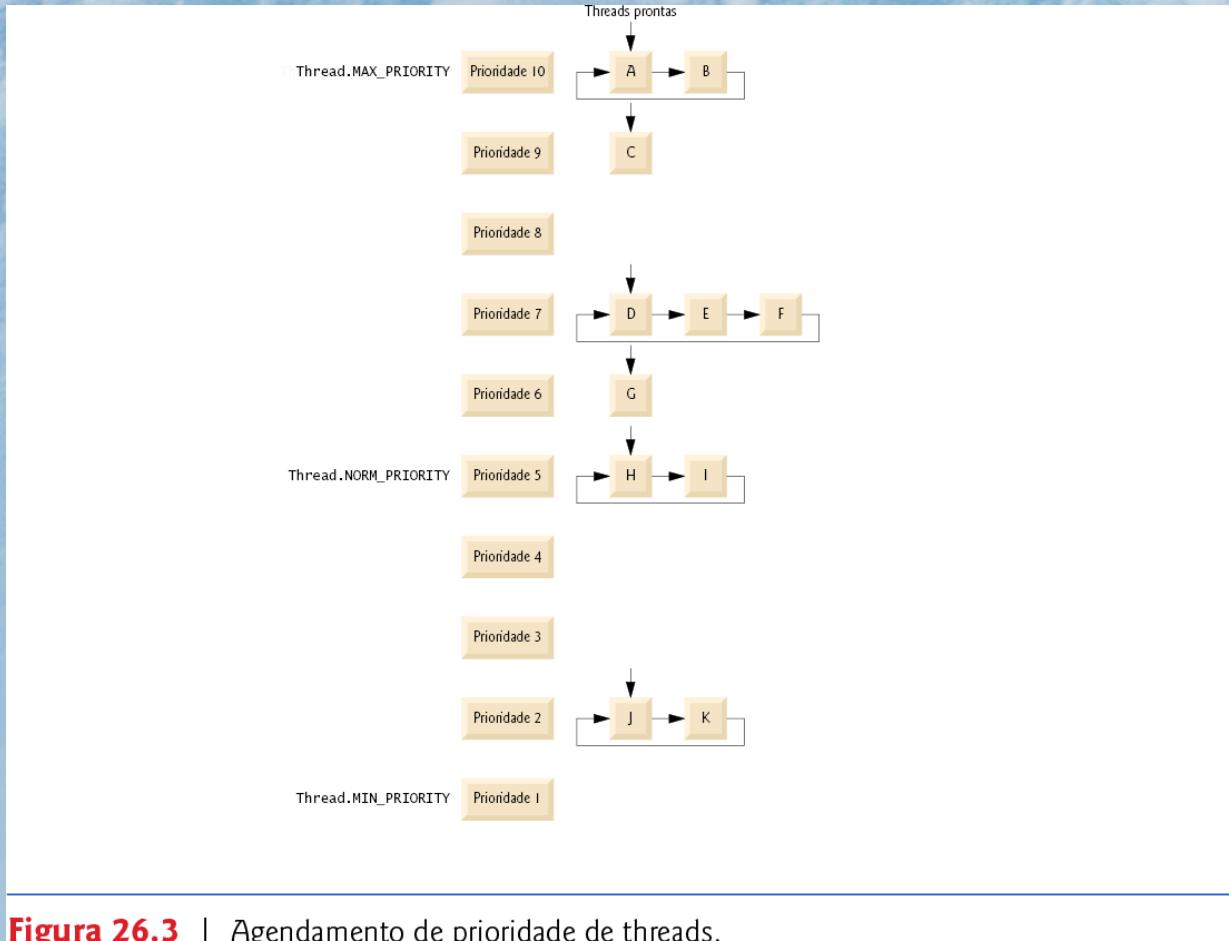
8^a edição

26.3 Prioridades de thread e agendamento de thread

- ▶ Toda thread do Java tem uma **prioridade de thread** que ajuda a determinar a ordem em que as threads são agendadas.
- ▶ As prioridades do Java variam entre `MIN_PRIORITY` (uma constante de 1) e `MAX_PRIORITY` (uma constante de 10). Por padrão, toda thread recebe a prioridade `NORM_PRIORITY` (uma constante de 5).
- ▶ Cada nova thread herda a prioridade da thread que a cria.

COMO PROGRAMAR

8ª edição





COMO PROGRAMAR

8^a edição



Dica de portabilidade 26.1

O agendamento de threads é dependente de plataforma — o comportamento de um programa de múltiplas threads pode variar nas diferentes implementações do Java.



COMO PROGRAMAR

8^a edição



Dica de portabilidade 26.2

Ao projetar programas de múltiplas threads, considere as capacidades de threading de todas as plataformas nas quais os programas executarão. Utilizar prioridades diferentes das prioridades padrão tornará a comportamento dos seus programas dependentes da plataforma. Se seu objetivo for a portabilidade, não ajuste as prioridades de thread.



COMO PROGRAMAR

8^a edição

26.4 Criando e executando threads

- ▶ Um objeto **Runnable** representa uma “tarefa” que pode executar concorrentemente com outras tarefas.
- ▶ O **Runnable** declara o método **run** único, que contém o código que define a tarefa que um objeto **Runnable** deve realizar.
- ▶ Quando uma thread executando um **Runnable** é criada e iniciada, ela chama o método **run** do objeto **Runnable**, que executa na nova thread.



COMO PROGRAMAR

8^a edição

26.4.1 Runnables e a classe Thread

- ▶ A classe `PrintTask` (Figura 26.4) implementa `Runnable` (linha 5), de modo que múltiplos `PrintTasks` possam executar concorrentemente.
- ▶ O método `sleep static` da thread coloca uma thread no *estado de espera sincronizado* pelo período de tempo especificado.
- ▶ Pode lançar uma exceção verificada do tipo `InterruptedException` se o método `interrupt` da thread adormecida for chamado.
- ▶ O código em `main` executa na **thread principal**, uma thread criada pela JVM.
- ▶ O código no método `run` de `PrintTask` executa nas threads criadas em `main`.
- ▶ Quando o método `main` termina, o programa em si continua executando porque ainda existem threads que são ativas.
- ▶ O programa não terminará até que sua última thread complete a execução.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.4: PrintTask.java
2 // Classe PrintTask dorme por um tempo aleatório de 0 a 5 segundos
3 import java.util.Random;
4
5 public class PrintTask implements Runnable
6 {
7     private final int sleepTime; // tempo de adormecimento aleatório para a thread
8     private final String taskName; // nome da tarefa
9     private final static Random generator = new Random();
10
11    public PrintTask( String name )
12    {
13        taskName = name; // configura o nome da tarefa
14
15        // seleciona tempo de adormecimento aleatório entre 0 e 5 segundos
16        sleepTime = generator.nextInt( 5000 ); // milissegundos
17    } // fim do construtor PrintTask
18
```

Figura 26.4 | A classe PrintTask dorme por um tempo aleatório de 0 a 5 segundos. (Parte 1 de 2.)

COMO PROGRAMAR

8ª edição

```
19 // método run contém o código que uma thread executará
20 public void run() ←
21 {
22     try // coloca a thread para dormir pelo período de tempo sleepTime
23     {
24         System.out.printf( "%s going to sleep for %d milliseconds.\n",
25             taskName, sleepTime );
26         Thread.sleep( sleepTime ); // coloca a thread para dormir
27     } // fim do try
28     catch ( InterruptedException exception )
29     {
30         System.out.printf( "%s %s\n", taskName,
31             "terminated prematurely due to interruption" );
32     } // fim do catch
33
34     // imprime o nome da tarefa
35     System.out.printf( "%s done sleeping\n", taskName );
36 } // fim do método run
37 } // fim da classe PrintTask
```

Os métodos chamados daqui executam
como parte da thread que chama run

Figura 26.4 | A classe PrintTask dorme por um tempo aleatório de 0 a 5 segundos. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.5: ThreadCreator.java
2 // Criando e iniciando três threads para executar Runnables.
3 import java.lang.Thread;
4
5 public class ThreadCreator
6 {
7     public static void main( String[] args )
8     {
9         System.out.println( "Creating threads" );
10
11         // cria cada thread com um novo runnable selecionado
12         Thread thread1 = new Thread( new PrintTask( "task1" ) );
13         Thread thread2 = new Thread( new PrintTask( "task2" ) );
14         Thread thread3 = new Thread( new PrintTask( "task3" ) );
15
16         System.out.println( "Threads created, starting tasks." );
17
18         // inicia threads e coloca no estado executável
19         thread1.start(); // invoca o método run de task1
20         thread2.start(); // invoca o método run de task2
21         thread3.start(); // invoca o método run de task3
22
23         System.out.println( "Tasks started, main ends.\n" ); ←
24     } // fim de main
25 } // fim da classe ThreadCreator
```

A thread principal finaliza depois disso, mas o programa continua até outras threads terminarem.

Figura 26.5 | Criando e iniciando três threads para executar Runnables. (Parte I de 2.)



COMO PROGRAMAR

8^a edição

```
Creating threads
Threads created, starting tasks
Tasks started, main ends
```

```
task3 going to sleep for 491 milliseconds
task2 going to sleep for 71 milliseconds
task1 going to sleep for 3549 milliseconds
task2 done sleeping
task3 done sleeping
task1 done sleeping
```

```
Creating threads
Threads created, starting tasks
task1 going to sleep for 4666 milliseconds
task2 going to sleep for 48 milliseconds
task3 going to sleep for 3924 milliseconds
Tasks started, main ends
```

```
task2 done sleeping
task3 done sleeping
task1 done sleeping
```

Figura 26.5 | Criando e iniciando três threads para executar Runnables. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

26.4.2 Gerenciamento de thread com o framework Executor

- ▶ É recomendado o uso da interface **Executor** para gerenciar a execução de objetos **Runnable** para você.
- ▶ Em geral, cria e gerencia um grupo de threads chamado de **pool de threads** para executar **Runnables**.
- ▶ **Executors** podem reusar threads existentes e podem aprimorar o desempenho otimizando o número de threads.
- ▶ O método **Executor execute** aceita um **Runnable** como um argumento.
- ▶ Um **Executor** atribui cada **Runnable** passado para seu método **execute** a uma das threads disponíveis no pool de threads.
- ▶ Se não houver threads disponíveis, o **Executor** cria uma nova thread ou espera uma thread tornar-se disponível.



COMO PROGRAMAR

8^a edição

- ▶ A interface **ExecutorService** estende **Executor** e declara métodos para gerenciar o ciclo de um **Executor**.
- ▶ Um objeto que implementa essa interface pode ser criado usando os métodos **static** declarados na classe **Executors**.
- ▶ O método **Executors newCachedThreadPool** retorna um **ExecutorService** que cria novas threads conforme a necessidade do aplicativo.
- ▶ O método **ExecutorService shutdown** notifica o **ExecutorService** para parar de aceitar novas tarefas, mas continua executando as tarefas que já foram enviadas.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.6: TaskExecutor.java
2 // Utilizando um ExecutorService para executar Runnables.
3 import java.util.concurrent.Executors;
4 import java.util.concurrent.ExecutorService;
5
6 public class TaskExecutor
7 {
8     public static void main( String[] args )
9     {
10         // cria e nomeia cada executável
11         PrintTask task1 = new PrintTask( "task1" );
12         PrintTask task2 = new PrintTask( "task2" );
13         PrintTask task3 = new PrintTask( "task3" );
14
15         System.out.println( "Starting Executor" );
16
17         // cria ExecutorService para gerenciar threads
18         ExecutorService threadExecutor = Executors.newCachedThreadPool();
19
20         // inicia threads e coloca no estado executável
21         threadExecutor.execute( task1 ); // inicia task1
22         threadExecutor.execute( task2 ); // inicia task2
23         threadExecutor.execute( task3 ); // inicia task3
```

Figura 26.6 | Utilizando um ExecutorService para executar Runnables. (Parte I de 3.)



COMO PROGRAMAR

8^a edição

```
24
25      // encerra threads trabalhadoras quando suas tarefas terminarem
26      threadExecutor.shutdown(); ←
27
28      System.out.println( "Tasks started, main ends.\n" );
29  } // fim de main
30 } // fim da classe TaskExecutor
```

Nenhuma thread pode ser iniciada com esse pool de threads nesse ponto

```
Starting Executor
Tasks started, main ends
```

```
task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
task3 done sleeping
task2 done sleeping
task1 done sleeping
```

Figura 26.6 | Utilizando um ExecutorService para executar Runnables. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
Starting Executor
```

```
task1 going to sleep for 1342 milliseconds
```

```
task2 going to sleep for 277 milliseconds
```

```
task3 going to sleep for 2737 milliseconds
```

```
Tasks started, main ends
```

```
task2 done sleeping
```

```
task1 done sleeping
```

```
task3 done sleeping
```

Figura 26.6 | Utilizando um ExecutorService para executar Runnables. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

26.5 Sincronização de thread

- ▶ Quando múltiplas threads compartilham um objeto e ele é modificado por uma ou várias delas, podem ocorrer resultados a menos que o acesso ao objeto compartilhado seja gerenciado apropriadamente.
- ▶ O problema pode ser resolvido fornecendo a uma única thread por vez o código de acesso *exclusivo* que manipula o objeto compartilhado.
- ▶ Durante esse tempo, outras threads que desejarem manipular o objeto são mantidas na espera.
- ▶ Quando a thread com acesso exclusivo ao objeto terminar de manipulá-lo, uma das threads que foi mantida na espera tem a permissão de prosseguir.
- ▶ Esse processo, chamado **sincronização de threads**, coordena o acesso a dados compartilhados por múltiplas threads concorrentes.
- ▶ Isso garante que uma thread que acessa um objeto compartilhado exclui todas as outras threads de fazer isso simultaneamente — isso é chamado de **exclusão mútua**.



COMO PROGRAMAR

8^a edição

- ▶ Uma maneira comum de realizar a sincronização é utilizar os **monitores** predefinidos do Java.
- ▶ Todo objeto tem um monitor e um **bloqueio de monitor** (ou **bloqueio intrínseco**).
- ▶ Permite armazenar no máximo apenas uma thread por vez.
- ▶ Uma thread deve adquirir o bloqueio antes de prosseguir com a operação.
- ▶ Outras threads tentando realizar uma operação que requer o mesmo bloqueio serão *bloqueadas*.
- ▶ Para especificar que uma thread deve manter um bloqueio de monitor para executar um bloco do código, o código deve ser colocado em uma **instrução synchronized**.
- ▶ Dizemos que está **guardada** por bloqueio de monitor.



COMO PROGRAMAR

8^a edição

- ▶ As instruções **synchronized** são declaradas utilizando a **palavra-chave synchronized**:
 - `synchronized (objeto)`
`{`
 instruções
`} // fim da instrução synchronized`
- ▶ onde *objeto* é o objeto cujo bloqueio de monitor será adquirido.
- ▶ *objeto* é normalmente **this** se ele for o objeto em que a instrução **synchronized** aparece.
- ▶ Quando uma instrução **synchronized** termina de executar, o bloqueio de monitor do objeto é liberado.
- ▶ O Java também permite **métodos synchronized**.



COMO PROGRAMAR

8^a edição

26.5.1 Compartilhamento de dados não sincronizados

- ▶ Um objeto `SimpleArray` (Figura 26.7) será compartilhado por múltiplas threads.
- ▶ Permitirá àquelas threads colocar valores `int` no `array`.
- ▶ A linha 26 coloca a thread que invoca `add` para dormir por um intervalo aleatório de 0 a 499 milissegundos.
- ▶ Isso é feito para tornar mais óbvios os problemas associados com o acesso não sincronizado a dados compartilhados.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.7: SimpleArray.java
2 // A classe que gerencia um array de inteiros a ser compartilhado por várias threads.
3 import java.util.Arrays;
4 import java.util.Random;
5
6 public class SimpleArray // ATENÇÃO: NÃO SEGURO PARA THREADS!
7 {
8     private final int[] array; // o array de inteiros compartilhado
9     private int writeIndex = 0; // o índice do próximo elemento a ser gravado
10    private final static Random generator = new Random();
11
12    // constrói um SimpleArray de um dado tamanho
13    public SimpleArray( int size )
14    {
15        array = new int[ size ];
16    } // fim do construtor
17
18    // adiciona um valor ao array compartilhado
19    public void add( int value )
20    {
21        int position = writeIndex; // armazena o índice de gravação
22    }
}
```

Figura 26.7 | A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads. (Parte I de 3.)



COMO PROGRAMAR

8^a edição

```
23     try
24     {
25         // coloca a thread para dormir por 0-499 milissegundos
26         Thread.sleep( generator.nextInt( 500 ) );
27     } // fim do try
28     catch ( InterruptedException ex )
29     {
30         ex.printStackTrace();
31     } // fim do catch
32
33     // coloca valor no elemento correto
34     array[ position ] = value;
35     System.out.printf( "%s wrote %2d to element %d.\n",
36                         Thread.currentThread().getName(), value, position );
37
38     ++writeIndex; // incrementa índice do elemento a ser gravado depois
39     System.out.printf( "Next write index: %d\n", writeIndex );
40 } // fim do método add
41
```

Figura 26.7 | A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
42     // utilizado para gerar saída do conteúdo do array de inteiros compartilhado
43     public String toString()
44     {
45         return "\nContents of SimpleArray:\n" + Arrays.toString( array );
46     } // fim do método toString
47 } // fim da classe SimpleArray
```

Figura 26.7 | A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

- ▶ A classe **ArrayWriter** (Figura 26.8) implementa a interface **Runnable** para definir uma tarefa para inserir valores em um objeto **SimpleArray**.
- ▶ A tarefa se completa depois que três números inteiros consecutivos que iniciam com **startValue** forem adicionados ao objeto **SimpleArray**.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.8: ArrayWriter.java
2 // Adiciona números inteiros a um array compartilhado com outros Runnables
3 import java.lang.Runnable;
4
5 public class ArrayWriter implements Runnable
6 {
7     private final SimpleArray sharedSimpleArray;
8     private final int startValue;
9
10    public ArrayWriter( int value, SimpleArray array )
11    {
12        startValue = value;
13        sharedSimpleArray = array;
14    } // fim do construtor
15
16    public void run()
17    {
18        for ( int i = startValue; i < startValue + 3; i++ )
19        {
20            sharedSimpleArray.add( i ); // adiciona um elemento ao array compartilhado
21        } // for final
22    } // fim do método run
23 } // fim da classe ArrayWriter
```

Figura 26.8 | Adiciona números inteiros a um array compartilhado com outros Runnables.



COMO PROGRAMAR

8^a edição

- ▶ A classe **SharedArrayTest** (Figura 26.9) executa duas tarefas **ArrayWriter** que adicionam valores a um objeto **SimpleArray** simples.
- ▶ O método **shutdown** do ExecutorService impede tarefas adicionais de iniciar e permitir ao aplicativo terminar quando tarefas em execução completarem a execução.
- ▶ Gostaríamos de gerar saída do objeto **SimpleArray** para mostrar os resultados *depois* de as threads completarem suas tarefas.
- ▶ Portanto, precisamos que o programa espere as threads completarem suas tarefas antes de **main** gerar saída do conteúdo do objeto **SimpleArray**.
- ▶ A interface **ExecutorService** fornece o método **awaitTermination** com essa finalidade — retorna controle ao seu chamador quando todas as tarefas em execução em **ExecutorService** completarem ou quando o tempo-limite especificado expirar.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.9: SharedArrayTest.java
2 // Executa dois Runnables para adicionar elementos a um SimpleArray compartilhado.
3 import java.util.concurrent.Executors;
4 import java.util.concurrent.ExecutorService;
5 import java.util.concurrent.TimeUnit;
6
7 public class SharedArrayTest
8 {
9     public static void main( String[] args )
10    {
11        // constrói o objeto compartilhado
12        SimpleArray sharedSimpleArray = new SimpleArray( 6 );
13
14        // cria duas tarefas a serem gravadas no SimpleArray compartilhado
15        ArrayWriter writer1 = new ArrayWriter( 1, sharedSimpleArray );
16        ArrayWriter writer2 = new ArrayWriter( 11, sharedSimpleArray );
17
18        // executa as tarefas com um ExecutorService
19        ExecutorService executor = Executors.newCachedThreadPool();
20        executor.execute( writer1 );
21        executor.execute( writer2 );
22
23        executor.shutdown();
```

Figura 26.9 | Executa dois Runnables para inserir valores em um array compartilhado. (Parte 1 de 3.)



COMO PROGRAMAR

8^a edição

```
24
25     try
26     {
27         // espera 1 minuto por ambos os gravadores terminarem a execução
28         boolean tasksEnded = executor.awaitTermination(
29             1, TimeUnit.MINUTES );
30
31         if ( tasksEnded )
32             System.out.println( sharedSimpleArray ); // imprime o conteúdo
33         else
34             System.out.println(
35                 "Timed out while waiting for tasks to finish." );
36     } // fim do try
37     catch ( InterruptedException ex )
38     {
39         System.out.println(
40             "Interrupted while waiting for tasks to finish." );
41     } // fim do catch
42     } // fim de main
43 } // fim da classe SharedArrayTest
```

Figura 26.9 | Executa dois Runnables para inserir valores em um array compartilhado. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
pool-1-thread-1 wrote 1 to element 0.  
Next write index: 1  
pool-1-thread-1 wrote 2 to element 1.  
Next write index: 2  
pool-1-thread-1 wrote 3 to element 2.  
Next write index: 3  
pool-1-thread-2 wrote 11 to element 0.  
Next write index: 4  
pool-1-thread-2 wrote 12 to element 4.  
Next write index: 5  
pool-1-thread-2 wrote 13 to element 5.  
Next write index: 6  
  
Contents of SimpleArray:  
[11, 2, 3, 0, 12, 13]
```

O primeiro pool-1-thread-1 grava o valor 1 no elemento 0. Depois, pool-1-thread-2 grava o valor 11 no elemento 0, sobrescrevendo assim o valor previamente armazenado.

Figura 26.9 | Executa dois Runnables para inserir valores em um array compartilhado. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

26.5.2 Compartilhamento de dados sincronizados — tornando operações atômicas

- ▶ Os erros de saída (Figura 26.9) podem ser atribuídos ao fato de que o objeto compartilhado, `SimpleArray`, não é **seguro para threads**.
- ▶ Se uma única thread obtiver o valor de `writeIndex`, não haverá garantia de que outra thread não possa avançar e incrementar `writeIndex` antes de a primeira thread ter tido uma chance de colocar um valor no array.
- ▶ Se isso acontecer, a primeira thread estará gravando no array com base em um **valor obsoleto** de `writeIndex` — um valor que não é mais válido.



COMO PROGRAMAR

8^a edição

- ▶ Uma **operação atômica** não pode ser dividida em suboperações menores.
- ▶ Podemos simular a atomicidade assegurando que apenas uma thread executa as três operações por vez.
- ▶ A atomicidade pode ser alcançada utilizando a palavra-chave **synchronized**.



COMO PROGRAMAR

8^a edição



Observação de engenharia de software 26.1

Coloque todos os acessos a dados mutáveis que podem ser compartilhados por múltiplas threads dentro de instruções synchronized ou métodos synchronized que sincronizam no mesmo bloqueio. Ao realizar múltiplas operações em dados compartilhados, mantenha o bloqueio pela totalidade da operação para assegurar que a operação seja efetivamente atômica.



COMO PROGRAMAR

8^a edição

- ▶ A Figura 26.10 exibe a classe **SimpleArray** com a sincronização apropriada.
- ▶ Idêntica à classe **SimpleArray** da Figura 26.7, exceto que **add** é agora um método **synchronized** (linha 20) — apenas uma thread por vez pode executar esse método.
- ▶ Reutilizamos as classes **ArrayWriter** (Figura 26.8) e **SharedArrayTest** (Fig. 26.9) do exemplo anterior.
- ▶ Geramos a saída de mensagens de blocos **synchronized** para propósitos de demonstração.
- ▶ A E/S *não* deve ser realizada em blocos **synchronized**, porque é importante minimizar o período de tempo em que um objeto é “bloqueado”.
- ▶ A linha 27 nesse exemplo chama o método **Thread sleep** para enfatizar a imprevisibilidade do agendamento de thread.
- ▶ *Nunca* chame **sleep** enquanto mantiver um bloqueio em um aplicativo real.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.10: SimpleArray.java
2 // A classe que gerencia um array de inteiros a ser compartilhado por várias threads
3 // threads with synchronization.
4 import java.util.Arrays;
5 import java.util.Random;
6
7 public class SimpleArray
8 {
9     private final int[] array; // o array de inteiros compartilhado
10    private int writeIndex = 0; // o índice do próximo elemento a ser gravado
11    private final static Random generator = new Random();
12
13    // constrói um SimpleArray de um dado tamanho
14    public SimpleArray( int size )
15    {
16        array = new int[ size ];
17    } // fim do construtor
18}
```

Figura 26.10 | A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads com a sincronização. (Parte 1 de 3.)



COMO PROGRAMAR

8^a edição

```
19     // adiciona um valor ao array compartilhado
20     public synchronized void add( int value ) ← Somente uma thread por vez pode
21     {
22         int position = writeIndex; // armazena o índice de gravação
23
24         try
25         {
26             // coloca a thread para dormir por 0-499 milissegundos
27             Thread.sleep( generator.nextInt( 500 ) );
28         } // fim do try
29         catch ( InterruptedException ex )
30         {
31             ex.printStackTrace();
32         } // fim do catch
33
34         // coloca valor no elemento correto
35         array[ position ] = value;
36         System.out.printf( "%s wrote %d to element %d.\n",
37             Thread.currentThread().getName(), value, position );
38
39         ++writeIndex; // incrementa índice do elemento a ser gravado depois
40         System.out.printf( "Next write index: %d\n", writeIndex );
41     } // fim do método add
```

Figura 26.10 | A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads com a sincronização. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
42      // utilizado para gerar saída do conteúdo do array de inteiros compartilhado
43      public String toString()
44      {
45          return "\nContents of SimpleArray:\n" + Arrays.toString( array );
46      } // fim do método toString
47  } // fim da classe SimpleArray
```

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-2 wrote 11 to element 1.
Next write index: 2
pool-1-thread-2 wrote 12 to element 2.
Next write index: 3
pool-1-thread-2 wrote 13 to element 3.
Next write index: 4
pool-1-thread-1 wrote  2 to element 4.
Next write index: 5
pool-1-thread-1 wrote  3 to element 5.
Next write index: 6

Contents of SimpleArray:
1 11 12 13 2 3
```

Figura 26.10 | A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads com a sincronização. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição



Dica de desempenho 26.2

Mantenha a duração de instruções synchronized o mais curta possível mantendo ao mesmo tempo a sincronização necessária. Isso minimiza o tempo de espera por threads bloqueadas. Evite realizar E/S, cálculos longos e operações que não exigem a sincronização enquanto mantiver um bloqueio.



COMO PROGRAMAR

8^a edição

- ▶ A sincronização é necessária somente para **dados mutáveis**, ou dados que podem mudar durante seu tempo de vida.
- ▶ Se os dados compartilhados não mudarem em um programa de múltiplas threads, então não é possível para uma thread ver valores antigos ou incorretos em consequência da manipulação desses dados por outra thread.
- ▶ Ao compartilhar dados imutáveis por threads, declare os campos de dados correspondentes **final** para indicar que os valores das variáveis não mudarão depois de eles serem inicializados.
- ▶ Evita a modificação accidental.
- ▶ Rotular referências de objeto como **final** indica que a referência não mudará, mas não garante que o próprio objeto seja imutável — isso depende inteiramente das propriedades do objeto.



COMO PROGRAMAR

8^a edição



Boa prática de programação 26.I

Sempre declare como final campos de dados que você não espera que mudem. As variáveis primitivas que são declaradas como final podem ser seguramente compartilhadas pelas threads. Uma referência de objeto que é declarada como final assegura que o objeto que ela referencia será totalmente construído e inicializado antes de ser utilizado pelo programa, e impede que a referência aponte para outro objeto.



COMO PROGRAMAR

8^a edição

26.6 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** do aplicativo lê os dados do objeto compartilhado.
- ▶ Uma **thread produtora** gera dados e os coloca em um objeto compartilhado chamado **buffer**.
- ▶ Uma **thread consumidora** lê dados do buffer.
- ▶ Esse relacionamento requer que a sincronização assegure que os valores sejam produzidos e consumidos corretamente.
- ▶ As operações nos dados de buffers compartilhados por uma thread consumidora e produtora são também **dependentes de estado** — as operações devem prosseguir somente se o buffer estiver no estado correto.
- ▶ Se o buffer estiver em um estado não cheio, o produtor pode produzir; se o buffer estiver em um estado não vazio, o consumidor pode consumir.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.11: Buffer.java
2 // Interface Buffer especifica métodos chamados por Producer e Consumer.
3 public interface Buffer
4 {
5     // coloca o valor int no Buffer
6     public void set( int value ) throws InterruptedException;
7
8     // retorna o valor int a partir do Buffer
9     public int get() throws InterruptedException;
10 } // fim da interface Buffer
```

Figura 26.11 | A interface buffers especifica os métodos chamados por Producer e Consumer.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.12: Producer.java
2 // Producer com um método run que insere os valores de 1 a 10 no buffer.
3 import java.util.Random;
4
5 public class Producer implements Runnable
6 {
7     private final static Random generator = new Random();
8     private final 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;
```

Figura 26.12 | Producer com o método run que insere os valores 1 a 10 no buffer. (Parte I de 2.)



COMO PROGRAMAR

8^a edição

```
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 ) ); // sono aleatório
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 as linhas 25 ou 26 são interrompidas, imprimem o rastreamento de pilha
31         catch ( InterruptedException exception )
32         {
33             exception.printStackTrace();
34         } // fim do catch
35     } // fim do for
36
37     System.out.println(
38         "Producer done producing\nTerminating Producer" );
39     } // fim do método run
40 } // fim da classe Producer
```

Figura 26.12 | Producer com o método run que insere os valores 1 a 10 no buffer. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.13: Consumer.java
2 // Consumer com um método run que itera, lendo 10 valores do buffer.
3 import java.util.Random;
4
5 public class Consumer implements Runnable
6 {
7     private final static Random generator = new Random();
8     private final 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 10 vezes e soma os valores
17    public void run()
18    {
19        int sum = 0;
```

Figura 26.13 | Consumer com um método run que itera, lendo 10 valores de buffer. (Parte I de 2.)



COMO PROGRAMAR

8^a edição

```
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 as linhas 26 ou 27 são interrompidas, imprimem o rastreamento de pilha
31        catch ( InterruptedException exception )
32        {
33            exception.printStackTrace();
34        } // fim do catch
35    } // for final
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 26.13 | Consumer com um método run que itera, lendo 10 valores de buffer. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.14: UnsynchronizedBuffer.java
2 // UnsynchronizedBuffer mantém o número inteiro compartilhado que é acessado por
3 // uma thread produtora e uma thread consumidora via métodos set e get.
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 ) throws InterruptedException
10    {
11        System.out.printf( "Producer writes\t%d", value );
12        buffer = value;
13    } // fim do método set
14
15    // retorna valor do buffer
16    public int get() throws InterruptedException
17    {
18        System.out.printf( "Consumer reads\t%d", buffer );
19        return buffer;
20    } // fim do método get
21 } // fim da classe UnsynchronizedBuffer
```

Figura 26.14 | UnsynchronizedBuffer mantém o número inteiro compartilhado que é acessado por uma thread produtora e uma consumidora por meio dos métodos set e get.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.15: SharedBufferTest.java
2 // Aplicativo com duas threads que manipulam um buffer não sincronizado.
3 import java.util.concurrent.ExecutorService;
4 import java.util.concurrent.Executors;
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.newCachedThreadPool();
12
13         // cria UnsynchronizedBuffer para armazenar ints
14         Buffer sharedLocation = new UnsynchronizedBuffer();
15
16         System.out.println(
17             "Action\t\tValue\tSum of Produced\tSum of Consumed" );
18         System.out.println(
19             "-----\t\t-----\t-----\t-----\n" );
```

Figura 26.15 | Aplicativo com duas threads que manipulam um buffer não sincronizado. (Parte I de 6.)



COMO PROGRAMAR

8^a edição

```
21      // executa Producer e Consumer, fornecendo-lhes acesso
22      // a sharedLocation
23      application.execute( new Producer( sharedLocation ) );
24      application.execute( new Consumer( sharedLocation ) );
25
26      application.shutdown(); // termina o aplicativo quando as tarefas terminam
27  } // fim de main
28 } // fim da classe SharedBufferTest
```

Figura 26.15 | Aplicativo com duas threads que manipulam um buffer não sincronizado. (Parte 2 de 6.)



COMO PROGRAMAR

8^a edição

Action	Value	Sum of Produced	Sum of Consumed
Producer writes	1	1	
Producer writes	2	3	—— 1 é perdido
Producer writes	3	6	—— 2 é perdido
Consumer reads	3		3
Producer writes	4	10	
Consumer reads	4		7
Producer writes	5	15	
Producer writes	6	21	—— 5 é perdido
Producer writes	7	28	—— 6 é perdido
Consumer reads	7		14
Consumer reads	7		21 —— 7 lido novamente
Producer writes	8	36	
Consumer reads	8		29
Consumer reads	8		37 —— 8 é lido novamente
Producer writes	9	45	
Producer writes	10	55	—— 9 é perdido
Producer done producing			
Terminating Producer			
Consumer reads	10	47	
Consumer reads	10	57	—— 10 lido novamente
Consumer reads	10	67	—— 10 lido novamente
Consumer reads	10	77	—— 10 lido novamente

Figura 26.15 | Aplicativo com duas threads que manipulam um buffer não sincronizado.
(Parte 3 de 6.)



COMO PROGRAMAR

8^a edição

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

Figura 26.15 | Aplicativo com duas threads que manipulam um buffer não sincronizado. (Parte 4 de 6.)



COMO PROGRAMAR

8^a edição

Action	Value	Sum of Produced	Sum of Consumed
Consumer reads	-1		-1 — lê -1 dados inválidos
Producer writes	1	1	
Consumer reads	1		0
Consumer reads	1		1 — I lido novamente
Consumer reads	1		2 — I lido novamente
Consumer reads	1		3 — I lido novamente
Consumer reads	1		4 — I lido novamente
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	—— 5 é perdido
Consumer reads	6		19
Consumer read values totaling 19			
Terminating Consumer			
Producer writes	7	28	—— 7 nunca lido
Producer writes	8	36	—— 8 nunca lido
Producer writes	9	45	—— 9 nunca lido
Producer writes	10	55	—— 10 nunca lido

Figura 26.15 | Aplicativo com duas threads que manipulam um buffer não sincronizado. (Parte 5 de 6.)



COMO PROGRAMAR

8^a edição

Producer done producing
Terminating Producer

Figura 26.15 | Aplicativo com duas threads que manipulam um buffer não sincronizado. (Parte 6 de 6.)



COMO PROGRAMAR

8^a edição



Dica de prevenção de erro 26.1

O acesso a um objeto compartilhado pelas threads concorrentes deve ser controlado cuidadosamente ou um programa pode produzir resultados incorretos.



COMO PROGRAMAR

8^a edição

26.7 Relacionamento entre produtor e consumidor: ArrayBlockingQueue

- ▶ Um modo de sincronizar as threads consumidora e produtora é utilizar as classes do pacote de concorrência do Java que encapsulam a sincronização para você.
- ▶ O Java inclui a classe **ArrayBlockingQueue** (do pacote `java.util.concurrent`) — uma classe de buffers totalmente implementada, segura para threads que implementa a interface **BlockingQueue**.
- ▶ Declara métodos **put** e **take**, equivalentes de bloqueio dos métodos `Queue offer` e `poll`, respectivamente.
- ▶ O método **put** coloca um elemento no fim da **BlockingQueue**, esperando se a fila estiver cheia.
- ▶ O método **take** removerá um elemento da cabeça da **BlockingQueue**, esperando se a fila estiver vazia.

COMO PROGRAMAR

8ª edição

```
1 // Figura 26.16: BlockingBuffer.java
2 // Criando um buffer sincronizado com um ArrayBlockingQueue.
3 import java.util.concurrent.ArrayBlockingQueue;
4
5 public class BlockingBuffer implements Buffer
6 {
7     private final ArrayBlockingQueue<Integer> buffer; // buffer compartilhado
8
9     public BlockingBuffer()
10    {
11         buffer = new ArrayBlockingQueue<Integer>( 1 );
12     } // fim do construtor BlockingBuffer
13
14     // coloca o valor no buffer
15     public void set( int value ) throws InterruptedException
16     {
17         buffer.put( value ); // coloca o valor no buffer
18         System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
19                           "Buffer cells occupied: ", buffer.size() );
20     } // fim do método set
21 }
```

Trata a sincronização automaticamente

Figura 26.16 | Criando um buffer sincronizado com um ArrayBlockingQueue. (Parte I de 2.)



COMO PROGRAMAR

8^a edição

```
22     // retorna valor do buffer
23     public int get() throws InterruptedException
24     {
25         int readValue = buffer.take(); // remove o valor do buffer
26         System.out.printf( "%s %2d\t%s%d\n", "Consumer reads ",
27             readValue, "Buffer cells occupied: ", buffer.size() );
28
29         return readValue;
30     } // fim do método get
31 } // fim da classe BlockingBuffer
```

Figura 26.16 | Criando um buffer sincronizado com um ArrayBlockingQueue. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.17: BlockingBufferTest.java
2 // Duas threads que manipulam corretamente um buffer de bloqueio
3 // implementa o relacionamento produtor/consumidor.
4 import java.util.concurrent.ExecutorService;
5 import java.util.concurrent.Executors;
6
7 public class BlockingBufferTest
8 {
9     public static void main( String[] args )
10    {
11        // cria novo pool de threads com duas threads
12        ExecutorService application = Executors.newCachedThreadPool();
13
14        // cria BlockingBuffer para armazenar ints
15        Buffer sharedLocation = new BlockingBuffer();
16
17        application.execute( new Producer( sharedLocation ) );
18        application.execute( new Consumer( sharedLocation ) );
19
20        application.shutdown();
21    } // fim de main
22 } // fim da classe BlockingBufferTest
```

Figura 26.17 | Duas threads que manipulam um buffer de bloqueio que corretamente implementa o relacionamento produtor/consumidor. (Parte 1 de 2.)



COMO PROGRAMAR

8^a edição

```
Producer writes 1      Buffer cells occupied: 1
Consumer reads 1      Buffer cells occupied: 0
Producer writes 2      Buffer cells occupied: 1
Consumer reads 2      Buffer cells occupied: 0
Producer writes 3      Buffer cells occupied: 1
Consumer reads 3      Buffer cells occupied: 0
Producer writes 4      Buffer cells occupied: 1
Consumer reads 4      Buffer cells occupied: 0
Producer writes 5      Buffer cells occupied: 1
Consumer reads 5      Buffer cells occupied: 0
Producer writes 6      Buffer cells occupied: 1
Consumer reads 6      Buffer cells occupied: 0
Producer writes 7      Buffer cells occupied: 1
Consumer reads 7      Buffer cells occupied: 0
Producer writes 8      Buffer cells occupied: 1
Consumer reads 8      Buffer cells occupied: 0
Producer writes 9      Buffer cells occupied: 1
Consumer reads 9      Buffer cells occupied: 0
Producer writes 10     Buffer cells occupied: 1
Producer done producing
Terminating Producer
Consumer reads 10     Buffer cells occupied: 0

Consumer read values totaling 55
Terminating Consumer
```

Figura 26.17 | Duas threads que manipulam um buffer de bloqueio que corretamente implementa o relacionamento produtor/consumidor. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

26.8 Relacionamento entre produtor e consumidor com sincronização

- ▶ Com propósitos educativos, agora explicamos como você mesmo pode implementar um buffer compartilhado utilizando a palavra-chave **synchronized** e os métodos da classe **Object**.
- ▶ O primeiro passo na sincronização de acesso ao buffer é implementar os métodos **get** e **set** como métodos **synchronized**.
- ▶ Isso exige que uma thread obtenha o bloqueio de monitor no objeto **Buffer** antes de tentar acessar os dados do buffer.



COMO PROGRAMAR

8^a edição

- ▶ Os métodos **Object wait**, **notify** e **notifyAll** podem ser usados com condições para fazer as threads esperarem quando elas não puderem realizar suas tarefas.
- ▶ Chamar o método **wait** de **Object** em um objeto **synchronized** libera seu bloqueio de monitor e coloca a thread chamadora no *estado de espera*.
- ▶ Chamar o método **Object notify** em um objeto **synchronized** permite que uma thread na espera transite para o *estado executável* novamente.
- ▶ Se uma thread chamar **notifyAll** no objeto **synchronized**, então todas as threads esperando pelo bloqueio de monitor tornam-se elegíveis para readquirir o bloqueio.



COMO PROGRAMAR

8^a edição



Erro comum de programação 26.1

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



COMO PROGRAMAR

8^a edição



Dica de prevenção de erro 26.2

É uma boa prática utilizar `notifyAll` para notificar threads em espera para tornarem-se executáveis. Fazer isso evita a possibilidade de o programa se esquecer das threads em espera, que, se não fosse por isso, morreriam de inanição.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.18: SynchronizedBuffer.java
2 // Sincronizando acesso a dados compartilhados
3 // com os métodos wait e notifyAll de Object.
4 public class SynchronizedBuffer implements Buffer
5 {
6     private int buffer = -1; // compartilhado pelas threads producer e consumer
7     private boolean occupied = false; // se o buffer estiver ocupado
8
9     // coloca o valor no buffer
10    public synchronized void set( int value ) throws InterruptedException
11    {
12        // enquanto não houver posições vazias, coloca a thread em estado de espera
13        while ( occupied )
14        {
15            // envia informações de thread e as de buffer para a saída, então espera
16            System.out.println( "Producer tries to write." );
17            displayState( "Buffer full. Producer waits." );
18            wait();
19        } // fim do while
20
21        buffer = value; // configura novo valor de buffer
22    }
```

Figura 26.18 | Sincronizando acesso aos dados compartilhados utilizando os métodos Object wait e notifyAll. (Parte I de 3.)



COMO PROGRAMAR

8^a edição

```
23 // indica que a produtora não pode armazenar outro valor
24 // até a consumidora recuperar valor atual de buffer
25 occupied = true;
26
27     displayState( "Producer writes " + buffer );
28
29     notifyAll(); // instrui thread(s) em espera a entrar no estado executável
30 } // fim do método set; libera bloqueio em SynchronizedBuffer
31
32 // retorna valor do buffer
33 public synchronized int get() throws InterruptedException
34 {
35     // enquanto os dados não são lidos, coloca thread em estado de espera
36     while ( !occupied )
37     {
38         // envia informações de thread e as de buffer para a saída, então espera
39         System.out.println( "Consumer tries to read." );
40         displayState( "Buffer empty. Consumer waits." );
41         wait();
42     } // fim do while
43 }
```

Figura 26.18 | Sincronizando acesso aos dados compartilhados utilizando os métodos Object wait e notifyAll. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
44     // indica que a produtora pode armazenar outro valor
45     // porque a consumidora acabou de recuperar o valor do buffer
46     occupied = false;
47
48     displayState( "Consumer reads " + buffer );
49
50     notifyAll(); // instrui thread(s) em espera a entrar no estado executável
51
52     return buffer;
53 } // fim do método get; libera bloqueio em SynchronizedBuffer
54
55 // exibe a operação atual e o estado de buffer
56 public void displayState( String operation )
57 {
58     System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
59                         occupied );
60 } // fim do método displayState
61 } // fim da classe SynchronizedBuffer
```

Figura 26.18 | Sincronizando acesso aos dados compartilhados utilizando os métodos Object wait e notifyAll. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

- ▶ O campo **occupied** da classe **SynchronizedBuffer** é usado para determinar se é a vez do **Producer** ou do **Consumer** realizar uma tarefa.
- ▶ Esse campo é utilizado nas expressões condicionais tanto no método **set** como no **get**.
- ▶ Se **occupied** for **false**, então o **buffer** está vazio, portanto, o **Consumer** não pode ler o valor do **buffer**, mas o **Producer** pode colocar o valor no **buffer**.
- ▶ Se **occupied** for **true**, o **Consumer** pode ler um valor do **buffer**, mas o **Producer** não pode colocar um valor no **buffer**.



COMO PROGRAMAR

8^a edição



Dica de prevenção de erro 26.3

Sempre invoque o método wait em um loop que testa a condição na qual a tarefa está esperando. É possível que uma thread entre novamente no estado executável (via uma thread no estado de espera sincronizada ou outra thread chamando notifyAll) antes de a condição ser satisfeita. Testar a condição novamente assegura que a thread não executará de maneira errada se tiver sido notificada anteriormente.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.19: SharedBufferTest2.java
2 // Duas threads que manipulam corretamente 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 um newCachedThreadPool
11         ExecutorService application = Executors.newCachedThreadPool();
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",
17             "Operation",
18             "Buffer", "Occupied", "-----", "-----\t\t-----" );
19
20         // executa as tarefas Producer e Consumer
21         application.execute( new Producer( sharedLocation ) );
22         application.execute( new Consumer( sharedLocation ) );
23
24         application.shutdown();
25     } // fim de main
26 } // fim da classe SharedBufferTest2
```

Figura 26.19 | Duas threads que manipulam corretamente um buffer sincronizado. (Parte I de 4.)



COMO PROGRAMAR

8^a edição

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
Producer writes 4	4	true
Producer tries to write. Buffer full. Producer waits.	4	true

Figura 26.19 | Duas threads que manipulam corretamente um buffer sincronizado. (Parte 2 de 4.)



COMO PROGRAMAR

8^a edição

Producer writes 4	4	true
Producer tries to write. Buffer full. Producer waits.	4	true
Consumer reads 4	4	false
Producer writes 5	5	true
Consumer reads 5	5	false
Producer writes 6	6	true
Producer tries to write. Buffer full. Producer waits.	6	true
Consumer reads 6	6	false
Producer writes 7	7	true
Producer tries to write. Buffer full. Producer waits.	7	true
Consumer reads 7	7	false
Producer writes 8	8	true
Consumer reads 8	8	false
Consumer tries to read. Buffer empty. Consumer waits.	8	false

Figura 26.19 | Duas threads que manipulam corretamente um buffer sincronizado. (3 de 4.)



COMO PROGRAMAR

8^a edição

Producer writes 9	9	true
Consumer reads 9	9	false
Consumer tries to read. Buffer empty. Consumer waits.	9	false
Producer writes 10	10	true
Consumer reads 10	10	false
Producer done producing Terminating Producer		
Consumer read values totaling 55 Terminating Consumer		

Figura 26.19 | Duas threads que manipulam corretamente um buffer sincronizado. (Parte 4 de 4.)



COMO PROGRAMAR

8^a edição

26.9 Relacionamento entre produtor consumidor: buffers limitados

- ▶ O programa na Seção 26.8 pode não ter um desempenho ótimo.
- ▶ Se as duas threads operarem em diferentes velocidades, uma delas gastará mais (ou a maior parte) de seu tempo na espera.
- ▶ 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 sobre as velocidades relativas das threads concorrentes.*
- ▶ Quando as threads esperam excessivamente, os programas tornam-se menos eficientes, os programas interativos 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 delimitado** que fornece um número fixo de células de buffer em que o **Producer** pode colocar valores e a partir do qual o **Consumer** pode recuperar esses valores.



COMO PROGRAMAR

8^a edição



Dica de desempenho 26.3

Mesmo ao utilizar um buffer delimitado, é possível que uma thread produtora pudesse preencher o buffer, o que forçaria a produtora 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, uma thread consumidora deve esperar até que a produtora produza outro valor. A chave para utilizar um buffer delimitado é otimizar o tamanho do buffer para minimizar a quantidade de tempo de espera da thread, sem desperdiçar espaço.



COMO PROGRAMAR

8^a edição

- ▶ O modo mais simples de implementar um buffer limitado é utilizar um **ArrayBlockingQueue** para o buffer para que *todos os detalhes de sincronização sejam tratados por você*.
- ▶ Isso pode ser feito reutilizando o exemplo da Seção 26.7 e simplesmente passando o tamanho desejado do buffer limitado no construtor **ArrayBlockingQueue**.



COMO PROGRAMAR

8^a edição

- ▶ Implementando seu próprio buffer limitado como um buffer circular.
- ▶ O programa na Figura 26.20 e Figura 26.21 demonstra um **Producer** e um **Consumer** acessando um buffer delimitado com sincronização.
- ▶ Implementamos o buffer limitado na classe **CircularBuffer** (Figura 26.20) como um **buffer circular** que grava e lê elementos de array na ordem, começando da primeira para a última célula.
- ▶ Quando uma thread **Producer** ou **Consumer** alcança o último elemento, ela retorna ao primeiro e começa a gravar ou ler, respectivamente, a partir daí.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.20: CircularBuffer.java
2 // Sincronizando acesso a um buffer limitado de três elementos compartilhado.
3 public class CircularBuffer implements Buffer
4 {
5     private final int[] buffer = { -1, -1, -1 }; // buffer compartilhado
6
7     private int occupiedCells = 0; // conta número de buffers utilizados
8     private int writeIndex = 0; // índice do próximo elemento em que gravar
9     private int readIndex = 0; // índice do próximo elemento a ler
10
11    // coloca o valor no buffer
12    public synchronized void set( int value ) throws InterruptedException
13    {
14        // espera até o buffer ter espaço disponível, então grava o valor;
15        // enquanto não houver posições vazias, põe a thread no estado bloqueado
16        while ( occupiedCells == buffer.length )
17        {
18            System.out.printf( "Buffer is full. Producer waits.\n" );
19            wait(); // espera até uma célula do buffer ser liberada
20        } // fim do while
21
22        buffer[ writeIndex ] = value; // configura novo valor de buffer
```

Figura 26.20 | Sincronizando acesso a um buffer limitado de três elementos compartilhado. (Parte 1 de 4.)



COMO PROGRAMAR

8^a edição

```
23
24     // atualiza índice de gravação circular
25     writeIndex = ( writeIndex + 1 ) % buffer.length;
26
27     ++occupiedCells; // mais uma célula do buffer está cheia
28     displayState( "Producer writes " + value );
29     notifyAll(); // notifica threads que estão esperando para ler a partir do buffer
30 } // fim do método set
31
32 // retorna valor do buffer
33 public synchronized int get() throws InterruptedException
34 {
35     // espera até que o buffer tenha dados, então lê o valor;
36     // enquanto os dados não são lidos, coloca thread em estado de espera
37     while ( occupiedCells == 0 )
38     {
39         System.out.printf( "Buffer is empty. Consumer waits.\n" );
40         wait(); // espera até que uma célula do buffer seja preenchida
41     } // fim do while
42
43     int readValue = buffer[ readIndex ]; // lê valor do buffer
44 }
```

Figura 26.20 | Sincronizando acesso a um buffer limitado de três elementos compartilhado. (Parte 2 de 4.)



COMO PROGRAMAR

8^a edição

```
45     // atualiza índice de leitura circular
46     readIndex = ( readIndex + 1 ) % buffer.length;
47
48     --occupiedCells; // algumas poucas células de buffers estão ocupadas
49     displayState( "Consumer reads " + readValue );
50     notifyAll(); // notifica threads que estão esperando para gravar no buffer
51
52     return readValue;
53 } // fim do método get
54
55 // exibe a operação atual e o estado de buffer
56 public void displayState( String operation )
57 {
58     // gera saída de operação e número de células de buffers ocupados
59     System.out.printf( "%s%s%d)\n%s",
60                         operation,
61                         " (buffer cells occupied: ", occupiedCells, "buffer cells: " );
62
63     for ( int value : buffer )
64         System.out.printf( " %2d ", value ); // gera a saída dos valores no buffer
65
66     System.out.print( "\n" );
```

Figura 26.20 | Sincronizando acesso a um buffer limitado de três elementos compartilhado. (Parte 3 de 4.)



COMO PROGRAMAR

8^a edição

```
67     for ( int i = 0; i < buffer.length; i++ )
68         System.out.print( "---- " );
69
70     System.out.print( "\n" );
71
72     for ( int i = 0; i < buffer.length; i++ )
73     {
74         if ( i == writeIndex && i == readIndex )
75             System.out.print( " WR" ); // índice de gravação e leitura
76         else if ( i == writeIndex )
77             System.out.print( " W " ); // só grava índice
78         else if ( i == readIndex )
79             System.out.print( " R " ); // só lê índice
80         else
81             System.out.print( " " ); // nenhum dos índices
82     } // for final
83
84     System.out.println( "\n" );
85 } // fim do método displayState
86 } // fim da classe CircularBuffer
```

Figura 26.20 | Sincronizando acesso a um buffer limitado de três elementos compartilhado. (Parte 4 de 4.)



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.21: CircularBufferTest.java
2 // Threads Producer e Consumer 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.newCachedThreadPool();
12
13         // cria CircularBuffer para armazenar ints
14         CircularBuffer sharedLocation = new CircularBuffer();
15
16         // exibe o estado inicial do CircularBuffer
17         sharedLocation.displayState( "Initial State" );
18
19         // executa as tarefas Producer e Consumer
20         application.execute( new Producer( sharedLocation ) );
21         application.execute( new Consumer( sharedLocation ) );
22 }
```

Figura 26.21 | Threads Producer e Consumer que manipulam um buffer circular. (Parte I de 6.)



COMO PROGRAMAR

8^a edição

```
23         application.shutdown();
24     } // fim de main
25 } // fim da classe CircularBufferTest
```

Initial State (buffer cells occupied: 0)
buffer cells: -1 -1 -1

WR

Producer writes 1 (buffer cells occupied: 1)
buffer cells: 1 -1 -1

R W

Consumer reads 1 (buffer cells occupied: 0)
buffer cells: 1 -1 -1

WR

Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
buffer cells: 1 2 -1

R W

Figura 26.21 | Threads Producer e Consumer que manipulam um buffer circular. (Parte 2 de 6.)



COMO PROGRAMAR

8^a edição

Consumer reads 2 (buffer cells occupied: 0)
buffer cells: 1 2 -1

WR

Producer writes 3 (buffer cells occupied: 1)
buffer cells: 1 2 3

W R

Consumer reads 3 (buffer cells occupied: 0)
buffer cells: 1 2 3

WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells: 4 2 3

R W

Producer writes 5 (buffer cells occupied: 2)
buffer cells: 4 5 3

R W

Figura 26.21 | Threads Producer e Consumer que manipulam um buffer circular. (Parte 3 de 6.)



COMO PROGRAMAR

8^a edição

Consumer reads 4 (buffer cells occupied: 1)

buffer cells: 4 5 3

 R W

Producer writes 6 (buffer cells occupied: 2)

buffer cells: 4 5 6

 W R

Producer writes 7 (buffer cells occupied: 3)

buffer cells: 7 5 6

 WR

Consumer reads 5 (buffer cells occupied: 2)

buffer cells: 7 5 6

 W R

Producer writes 8 (buffer cells occupied: 3)

buffer cells: 7 8 6

 WR

Figura 26.21 | Threads Producer e Consumer que manipulam um buffer circular. (Parte 4 de 6.)



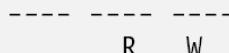
COMO PROGRAMAR

8^a edição

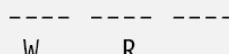
Consumer reads 6 (buffer cells occupied: 2)
buffer cells: 7 8 6



Consumer reads 7 (buffer cells occupied: 1)
buffer cells: 7 8 6



Producer writes 9 (buffer cells occupied: 2)
buffer cells: 7 8 9



Consumer reads 8 (buffer cells occupied: 1)
buffer cells: 7 8 9



Consumer reads 9 (buffer cells occupied: 0)
buffer cells: 7 8 9



Figura 26.21 | Threads Producer e Consumer que manipulam um buffer circular. (Parte 5 de 6.)



COMO PROGRAMAR

8^a edição

```
Producer writes 10 (buffer cells occupied: 1)
buffer cells:   10   8   9
```

 R W

```
Producer done producing
```

```
Terminating Producer
```

```
Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10   8   9
```

 WR

```
Consumer read values totaling: 55
```

```
Terminating Consumer
```

Figura 26.21 | Threads Producer e Consumer que manipulam um buffer circular. (Parte 6 de 6.)



COMO PROGRAMAR

8^a edição

26.10 Relacionamento entre produtor e consumidor: as interfaces Lock e Condition

- ▶ Nesta seção, discutimos as interfaces **Lock** e **Condition**, que foram introduzidas no Java SE 5.
- ▶ Essas interfaces permitem controle mais exato sobre a sincronização de threads, mas são mais complicadas de usar.
- ▶ Qualquer objeto pode conter uma referência para um objeto que implementa a interface **Lock** (do pacote `java.util.concurrent.locks`).
- ▶ Uma thread chama o método **lock** de **Lock** para adquirir o bloqueio.
- ▶ Assim que um **Lock** tiver sido obtido por uma thread, o objeto **Lock** não permitirá que outra thread o obtenha até a primeira thread liberá-lo (chamando o método **unlock** de **Lock**).
- ▶ Se várias threads estiverem tentando chamar o método **lock** no mesmo objeto **Lock** simultaneamente, apenas uma dessas threads poderá obter o bloqueio — todas as outras serão colocadas no estado de *espera* desse bloqueio.



COMO PROGRAMAR

8^a edição

- ▶ Quando uma thread chama o método `unlock`, o bloqueio do objeto é liberado e uma thread na espera tentando bloquear o objeto prossegue.
- ▶ A classe **ReentrantLock** (do 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 o argumento for `true`, a diretiva de imparcialidade do **ReentrantLock** é “a thread em espera mais longa que irá adquirir o bloqueio quando ele estiver disponível”.



COMO PROGRAMAR

8^a edição



Observação de engenharia de software 26.2

Utilizar um ReentrantLock com uma diretiva de imparcialidade evita o adiamento indefinido.



COMO PROGRAMAR

8^a edição



Dica de desempenho 26.4

Utilizar um ReentrantLock com uma diretiva de imparcialidade pode reduzir muito o desempenho de programa.



COMO PROGRAMAR

8^a edição

- ▶ Se uma thread que possui um **Lock** determina que não é possível continuar sua tarefa até que alguma condição seja satisfeita, a thread pode esperar em um **objeto condição**.
- ▶ Permite declarar explicitamente os objetos condição nos quais uma thread pode precisar esperar.
- ▶ Por exemplo, no relacionamento produtor/consumidor, os produtores podem esperar em um único objeto e os consumidores podem esperar em outro.
- ▶ Não é possível ao utilizar palavras-chave **synchronized** e o bloqueio de monitor predefinido de um objeto.
- ▶ Os objetos condição estão associados com um **Lock** específico e são criados chamando o método **newCondition** de **Lock**, que retorna um objeto que implementa a interface **Condition** (do pacote **java.util.concurrent.locks**).



COMO PROGRAMAR

8^a edição

- ▶ Para esperar um objeto 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**.
- ▶ Quando uma thread *executável* completa uma tarefa e determina que a thread de *espera* pode agora continuar, a thread *executável* pode chamar o método **Condition signal** para permitir que as nossas threads esperem o *estado de espera* de **Condition** retornem ao *estado executável*.
- ▶ Se uma thread chamar o método **Condition signalAll**, então todas as threads que esperam essa condição mudam para o *estado executável* e tornam-se adequadas para readquirir **Lock**.



COMO PROGRAMAR

8^a edição



Erro comum de programação 26.2

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, enquanto simultaneamente a *thread2* não pode prosseguir porque está esperando (direta ou indiretamente) a *thread1* prosseguir. As duas threads estão esperando uma à outra, então as ações que permitiriam a cada thread continuar a execução podem jamais ocorrer.



COMO PROGRAMAR

8^a edição



Dica de prevenção de erro 26.4

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 um objeto condição, uma thread separada, por fim, chame o método `Condition signal` para fazer a transição da thread em espera pelo objeto condição de volta para o estado executável. Se múltiplas threads podem estar esperando o objeto 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, poderia ocorrer inanição.



COMO PROGRAMAR

8^a edição



Erro comum de programação 26.3

Um `IllegalMonitorStateException` ocorre se uma thread executar um `await`, um `signal` ou um `signalAll` em um objeto `Condition` criado a partir de um `ReentrantLock` sem ter adquirido o bloqueio para esses objetos `Condition`.



COMO PROGRAMAR

8^a edição

- ▶ Locks permitem interromper threads em espera ou especificar um tempo-limite de espera para adquirir um bloqueio, que não é possível usando a palavra-chave **synchronized**.
- ▶ Além disso, um **Lock** não é forçado a ser adquirido e liberado no mesmo bloco do código.
- ▶ Objetos **Condition** permitem especificar múltiplas condições nas quais a thread podem esperar.
- ▶ Com a palavra-chave **synchronized**, não há nenhum modo de afirmar explicitamente a condição que as threads estão esperando.



COMO PROGRAMAR

8^a edição



Dica de prevenção de erro 26.5

O uso das interfaces Lock e Condition é propenso a erros — não é garantido que unlock será chamado, ao passo que o monitor em uma instrução synchronized sempre será liberado quando a instrução completar a execução.



COMO PROGRAMAR

8^a edição

- ▶ As linhas 14–15 criam duas **Conditions** usando o método **Lock newCondition**.
- ▶ **Condition canwrite** contém uma fila para uma thread **Producer** na espera enquanto o buffer estiver cheio.
- ▶ Se o buffer estiver cheio, o **Producer** chama o método **await** nessa **Condition**.
- ▶ Quando **Consumer** lê os dados de um buffer cheio, ele chama o método **signal** nessa **Condition**.
- ▶ **Condition canRead** contém uma fila para uma thread **Consumer** na espera enquanto o buffer estiver vazio (isto é, não há dados no buffer para o **Consumer** ler).
- ▶ Se o buffer estiver cheio, o **Consumer** chama o método **await** nessa **Condition**.
- ▶ Quando **Producer** grava os dados em um buffer vazio, ele chama o método **signal** nessa **Condition**.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.22: SynchronizedBuffer.java
2 // Sincronizando acesso a um número inteiro compartilhado com
3 // as interfaces Lock e Condition
4 import java.util.concurrent.locks.Lock;
5 import java.util.concurrent.locks.ReentrantLock;
6 import java.util.concurrent.locks.Condition;
7
8 public class SynchronizedBuffer implements Buffer
9 {
10     // Bloqueio para controlar sincronização com esse buffer
11     private final Lock accessLock = new ReentrantLock();
12
13     // condições para controlar leitura e gravação
14     private final Condition canWrite = accessLock.newCondition();
15     private final Condition canRead = accessLock.newCondition();
16
17     private int buffer = -1; // compartilhado pelas threads producer e consumer
18     private boolean occupied = false; // se o buffer estiver ocupado
19
20     // coloca o valor int no buffer
21     public void set( int value ) throws InterruptedException
22     {
```

Figura 26.22 | Sincronizando acesso a um número inteiro compartilhado com as interfaces Lock e Condition. (Parte 1 de 5.)



COMO PROGRAMAR

8^a edição

```
23     accessLock.lock(); // bloqueia esse objeto
24
25     // envia informações da thread e buffer para a saída, então espera
26     try
27     {
28         // enquanto o buffer não estiver vazio, coloca thread no estado de espera
29         while ( occupied )
30         {
31             System.out.println( "Producer tries to write." );
32             displayState( "Buffer full. Producer waits." );
33             canWrite.await(); // espera até que o buffer esteja vazio
34         } // fim do while
35
36         buffer = value; // configura novo valor de buffer
37
38         // indica que a produtora não pode armazenar outro valor
39         // até a consumidora recuperar o valor atual de buffer
40         occupied = true;
41
42         displayState( "Producer writes " + buffer );
43     }
```

Figura 26.22 | Sincronizando acesso a um número inteiro compartilhado com as interfaces Lock e Condition. (Parte 2 de 5.)



COMO PROGRAMAR

8^a edição

```
44         // sinaliza a thread que está esperando para ler a partir do buffer
45         canRead.signal();
46     } // fim do try
47     finally
48     {
49         accessLock.unlock(); // desbloqueia esse objeto
50     } // fim de finally
51 } // fim do método set
52
53 // retorna o valor do buffer
54 public int get() throws InterruptedException
55 {
56     int readValue = 0; // inicializa o valor lido a partir do buffer
57     accessLock.lock(); // bloqueia esse objeto
58
59     // envia informações de thread e buffer para a saída, então espera
60     try
61     {
62         // se não houver dados para serem lidos, coloca a thread em estado de espera
63         while ( !occupied )
64         {
65             System.out.println( "Consumer tries to read." );
```

Figura 26.22 | Sincronizando acesso a um número inteiro compartilhado com as interfaces Lock e Condition. (Parte 3 de 5.)



COMO PROGRAMAR

8^a edição

```
66         displayState( "Buffer empty. Consumer waits." );
67         canRead.await(); // espera até o buffer tornar-se cheio
68     } // fim do while
69
70     // indica que a produtora pode armazenar outro valor
71     // porque a consumidora acabou de recuperar o valor do buffer
72     occupied = false;
73
74     readValue = buffer; // recupera o valor do buffer
75     displayState( "Consumer reads " + readValue );
76
77     // sinaliza a thread que está esperando o buffer tornar-se vazio
78     canWrite.signal();
79 } // fim do try
80 finally
81 {
82     accessLock.unlock(); // desbloqueia esse objeto
83 } // fim de finally
84
85     return readValue;
86 } // fim do método get
87
```

Figura 26.22 | Sincronizando acesso a um número inteiro compartilhado com as interfaces Lock e Condition. (Parte 4 de 5.)



COMO PROGRAMAR

8^a edição

```
88     // exibe a operação atual e o estado de buffer
89     public void displayState( String operation )
90     {
91         System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
92                             occupied );
93     } // fim do método displayState
94 } // fim da classe SynchronizedBuffer
```

Figura 26.22 | Sincronizando acesso a um número inteiro compartilhado com as interfaces Lock e Condition. (Parte 5 de 5.)



COMO PROGRAMAR

8^a edição



Dica de prevenção de erro 26.6

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



COMO PROGRAMAR

8^a edição



Erro comum de programação 26.4

Esquecer de sinalizar uma thread em espera é um erro de lógica. A thread permanecerá no estado de espera, o que a impedirá de prosseguir. Tal espera pode levar ao adiamento indefinido ou impasse.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.23: SharedBufferTest2.java
2 // 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.newCachedThreadPool();
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",
17             "Operation",
18             "Buffer", "Occupied", "-----", "-----\t\t-----" );
19
20         // executa as tarefas Producer e Consumer
21         application.execute( new Producer( sharedLocation ) );
22         application.execute( new Consumer( sharedLocation ) );
23
24         application.shutdown();
25     } // fim de main
26 } // fim da classe SharedBufferTest2
```

Figura 26.23 | Duas threads que manipulam um buffer sincronizado. (Parte 1 de 4.)



COMO PROGRAMAR

8^a edição

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

Figura 26.23 | Duas threads que manipulam um buffer sincronizado. (Parte 2 de 4.)



COMO PROGRAMAR

8^a edição

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
Consumer reads 9	9	false

Figura 26.23 | Duas threads que manipulam um buffer sincronizado. (Parte 3 de 4.)



COMO PROGRAMAR

8^a edição

Producer writes 10

10

true

Producer done producing

Terminating Producer

Consumer reads 10

10

false

Consumer read values totaling 55

Terminating Consumer

Figura 26.23 | Duas threads que manipulam um buffer sincronizado. (Parte 4 de 4.)



COMO PROGRAMAR

8^a edição

26.11 Multithreading com GUI

- ▶ Todos os aplicativos Swing têm uma única thread, chamada **thread de despacho de eventos**, para tratar interações com os componentes GUI do aplicativo.
- ▶ Todas as tarefas que exigem interação com a GUI de um aplicativo são colocadas em uma fila de eventos e executadas em sequência pela thread de despacho de eventos.
- ▶ Os componentes GUI Swing não são seguros para thread.
- ▶ A segurança de thread em aplicativos GUI é alcançada assegurando que os componentes Swing são acessados apenas de uma única thread — a thread de despacho de eventos.
- ▶ Chamada de **confinamento de thread**.



COMO PROGRAMAR

8^a edição

- ▶ Java SE 6 fornece a classe **SwingWorker** (no pacote `javax.swing`) para realizar cálculos demorados em uma thread trabalhadora e atualizar componentes Swing a partir da thread de despacho de eventos com base nos resultados dos cálculos.
- ▶ Implementa a interface **Runnable**, o que significa que um objeto **Swingworker** pode ser agendado para executar em uma thread separada.
- ▶ Alguns métodos **Swingworker** comuns são descritos na Figura 26.24.



COMO PROGRAMAR

8^a edição

Método	Descrição
doInBackground	Define um cálculo longo e é chamado em uma thread trabalhadora.
done	Executa na thread de despacho de eventos quando doInBackground retorna.
execute	Agenda o objeto SwingWorker a ser executado em uma thread trabalhadora.
get	Espera o cálculo ser concluído e retorna o resultado do cálculo (isto é, o valor de retorno de doInBackground).
publish	Envia resultados intermediários a partir do método doInBackground para o método process para processamento na thread de despacho de eventos.
process	Recebe resultados intermediários a partir do método publish e os processa na thread de despacho de eventos.
setProgress	Configura a propriedade de progresso para notificar todos os ouvintes de alteração de propriedade na thread de despacho de eventos de atualizações de barra de progresso.

Figura 26.24 | Métodos SwingWorker comumente utilizados.



COMO PROGRAMAR

8^a edição

26.11.1 Realizando cálculos em um thread worker

- ▶ A classe `BackgroundCalculator` (Figura 26.25) estende `Swingworker` (linha 8), sobrescrevendo os métodos `doInBackground` e `done`.
- ▶ O método `doInBackground` (linhas 21–25) calcula o enésimo número de Fibonacci em uma thread trabalhadora e retorna o resultado.
- ▶ O método `done` (linhas 28–44) exibe o resultado em um `JLabel`.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.25: BackgroundCalculator.java
2 // Subclasse SwingWorker para calcular números de Fibonacci
3 // em uma thread de segundo plano.
4 import javax.swing.SwingWorker;
5 import javax.swing.JLabel;
6 import java.util.concurrent.ExecutionException;
7
8 public class BackgroundCalculator extends SwingWorker< Long, Object >
9 {
10     private final int n; // Número de Fibonacci a calcular
11     private final JLabel resultJLabel; // JLabel para exibir o resultado
12
13     // construtor
14     public BackgroundCalculator( int number, JLabel label )
15     {
16         n = number;
17         resultJLabel = label;
18     } // fim do construtor BackgroundCalculator
19
20     // código demorado a ser executado em uma thread trabalhadora
21     public Long doInBackground()
22     {
```

Figura 26.25 | Subclasse SwingWorker para calcular números de Fibonacci em uma thread em segundo plano. (Parte 1 de 3.)



COMO PROGRAMAR

8^a edição

```
23     long nthFib = fibonacci( n );
24     return String.valueOf( nthFib );
25 } // fim do método doInBackground
26
27 // código a executar na thread de despacho de eventos quando doInBackground retorna
28 protected void done()
29 {
30     try
31     {
32         // obtém o resultado de doInBackground e exibe-o
33         resultJLabel.setText( get().toString() );
34     } // fim do try
35     catch ( InterruptedException ex )
36     {
37         resultJLabel.setText( "Interrupted while waiting for results." );
38     } // fim do catch
39     catch ( ExecutionException ex )
40     {
41         resultJLabel.setText(
42             "Error encountered while performing calculation." );
43     } // fim do catch
44 } // fim do método done
```

Figura 26.25 | Subclasse SwingWorker para calcular números de Fibonacci em uma thread em segundo plano. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
45
46     // método recursivo fibonacci; calcula o enésimo número de Fibonacci
47     public long fibonacci( long number )
48     {
49         if ( number == 0 || number == 1 )
50             return number;
51         else
52             return fibonacci( number - 1 ) + fibonacci( number - 2 );
53     } // fim do método fibonacci
54 } // fim da classe BackgroundCalculator
```

Figura 26.25 | Subclasse SwingWorker para calcular números de Fibonacci em uma thread em segundo plano. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

- ▶ **Swingworker** é uma classe genérica.
- ▶ O primeiro parâmetro de tipo indica o tipo retornado pelo método **doInBackground**; o segundo indica o tipo que é passado entre os métodos **publish** e **process** para tratar dos resultados intermediários.
- ▶ Como não usamos **publish** e **process** nesse exemplo, basta utilizarmos **Object** como o segundo parâmetro de tipo.
- ▶ Quando o método **execute** é chamado um objeto **Backgroundcalculator**, o objeto é agendado para a execução em uma thread trabalhadora.
- ▶ O método **doInBackground** é chamado a partir da thread trabalhadora e invoca o método **fibonacci** (linhas 47–53), passando a variável de instância **n** como um argumento (linha 23).
- ▶ Quando **fibonacci** retorna, o método **doInBackground** retorna o resultado.



COMO PROGRAMAR

8^a edição

- ▶ Depois de `doInBackground` retornar, o método `done` é chamado a partir da thread de despacho de eventos.
- ▶ Esse método tenta configurar o resultado `JLabel` com o valor de retorno de `doInBackground` chamando o método `get` para recuperar o valor de retorno.
- ▶ Se necessário, o método `get` espera que o resultado esteja pronto, mas como o chamamos a partir do método `done`, o cálculo estará completo antes de `get` ser chamado.



COMO PROGRAMAR

8^a edição



Observação de engenharia de software 26.3

Todos os componentes GUI que forem manipulados pelos métodos SwingWorker, como os componentes que serão atualizados a partir dos métodos process ou done, devem ser passados para o construtor da subclasse SwingWorker e armazenados no objeto de subclasse. Isso dá a esses métodos acesso aos componentes GUI que eles manipularão.



COMO PROGRAMAR

8^a edição

- ▶ A classe **FibonacciNumbers** (Figura 26.26) exibe uma janela que contém dois conjuntos de componentes GUI — um configurado para calcular um número de Fibonacci em uma thread trabalhadora e o outro para obter o próximo número de Fibonacci em resposta ao clique de usuário em um **JButton**.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.26: FibonacciNumbers.java
2 // Utilizando SwingWorker para fazer um cálculo longo com
3 // resultados exibidos em uma GUI.
4 import java.awt.GridLayout;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.ActionListener;
7 import javax.swing.JButton;
8 import javax.swing.JFrame;
9 import javax.swing.JPanel;
10 import javax.swing.JLabel;
11 import javax.swing.JTextField;
12 import javax.swing.border.TitledBorder;
13 import javax.swing.border.LineBorder;
14 import java.awt.Color;
15 import java.util.concurrent.ExecutionException;
16
17 public class FibonacciNumbers extends JFrame
18 {
19     // componentes para calcular o Fibonacci de um número inserido pelo usuário
20     private final JPanel workerJPanel =
21         new JPanel( new GridLayout( 2, 2, 5, 5 ) );
22     private final JTextField numberJTextField = new JTextField();
```

Figura 26.26 | Utilizando SwingWorker para fazer um cálculo longo exibindo os resultados em uma GUI.
(Parte 1 de 7.)



COMO PROGRAMAR

8^a edição

```
23     private final JButton goJButton = new JButton( "Go" );
24     private final JLabel fibonacciJLabel = new JLabel();
25
26     // componentes e variáveis para obter o próximo número de Fibonacci
27     private final JPanel eventThreadJPanel =
28         new JPanel( new GridLayout( 2, 2, 5, 5 ) );
29     private long n1 = 0; // inicializa com o primeiro número de Fibonacci
30     private long n2 = 1; // inicializa com o segundo número de Fibonacci
31     private int count = 1; // número de Fibonacci atual a exibir
32     private final JLabel nJLabel = new JLabel( "Fibonacci of 1: " );
33     private final JLabel nFibonacciJLabel =
34         new JLabel( String.valueOf( n2 ) );
35     private final JButton nextNumberJButton = new JButton( "Next Number" );
36
37     // construtor
38     public FibonacciNumbers()
39     {
40         super( "Fibonacci Numbers" );
41         setLayout( new GridLayout( 2, 1, 10, 10 ) );
42
43         // adiciona componentes GUI ao painel SwingWorker
44         workerJPanel.setBorder( new TitledBorder(
45             new LineBorder( Color.BLACK ), "With SwingWorker" ) );
```

Figura 26.26 | Utilizando SwingWorker para fazer um cálculo longo exibindo os resultados em uma GUI.
(Parte 2 de 7.)



COMO PROGRAMAR

8^a edição

```
46    workerJPanel.add( new JLabel( "Get Fibonacci of:" ) );
47    workerJPanel.add( numberJTextField );
48    goJButton.addActionListener(
49        new ActionListener()
50    {
51        public void actionPerformed( ActionEvent event )
52        {
53            int n;
54
55            try
56            {
57                // recupera a entrada de usuário como um número inteiro
58                n = Integer.parseInt( numberJTextField.getText() );
59            } // fim do try
60            catch( NumberFormatException ex )
61            {
62                // exibe uma mensagem de erro se o usuário não
63                // inseriu um número inteiro
64                fibonacciJLabel.setText( "Enter an integer." );
65                return;
66            } // fim do catch
67        }
68    }
```

Figura 26.26 | Utilizando SwingWorker para fazer um cálculo longo exibindo os resultados em uma GUI.
(Parte 3 de 7.)



COMO PROGRAMAR

8^a edição

```
68      // indica que o cálculo iniciou
69      fibonacciJLabel.setText( "Calculating..." );
70
71      // cria uma tarefa para fazer o cálculo em segundo plano
72      BackgroundCalculator task =
73          new BackgroundCalculator( n, fibonacciJLabel );
74      task.execute(); // executa a tarefa
75  } // fim do método actionPerformed
76 } // fim da classe interna anônima
77 ); // fim da chamada para addActionListener
78 workerJPanel.add( goJButton );
79 workerJPanel.add( fibonacciJLabel );
80
81 // adiciona componentes GUI ao painel da thread de despacho de eventos
82 eventThreadJPanel.setBorder( new TitledBorder(
83     new LineBorder( Color.BLACK ), "Without SwingWorker" ) );
84 eventThreadJPanel.add( nJLabel );
85 eventThreadJPanel.add( nFibonacciJLabel );
86 nextNumberJButton.addActionListener(
87     new ActionListener()
88     {
89         public void actionPerformed( ActionEvent event )
90         {
```

Figura 26.26 | Utilizando SwingWorker para fazer um cálculo longo exibindo os resultados em uma GUI.
(Parte 4 de 7.)



COMO PROGRAMAR

8^a edição

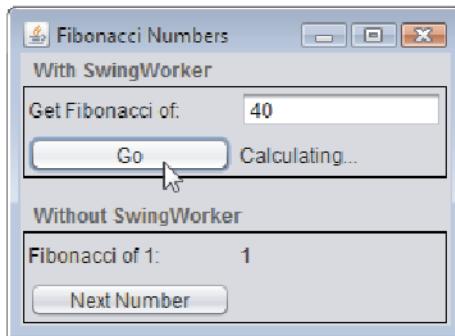
```
91         // calcula o número de Fibonacci depois de n2
92         long temp = n1 + n2;
93         n1 = n2;
94         n2 = temp;
95         ++count;
96
97         // exibe o seguinte número de Fibonacci
98         nJLabel.setText( "Fibonacci of " + count + ": " );
99         nFibonacciJLabel.setText( String.valueOf( n2 ) );
100     } // fim do método actionPerformed
101 } // fim da classe interna anônima
102 ); // fim da chamada para addActionListener
103 eventThreadJPanel.add( nextNumberJButton );
104
105 add( workerJPanel );
106 add( eventThreadJPanel );
107 setSize( 275, 200 );
108 setVisible( true );
109 } // fim do construtor
110
```

Figura 26.26 | Utilizando SwingWorker para fazer um cálculo longo exibindo os resultados em uma GUI.
(Parte 5 de 7.)

COMO PROGRAMAR

8ª edição

- a) Começa a calcular uma sequência Fibonacci de 40 números no segundo plano.



- b) Calculando outras sequências de valores Fibonacci enquanto a Fibonacci de 40 números continua calculando.

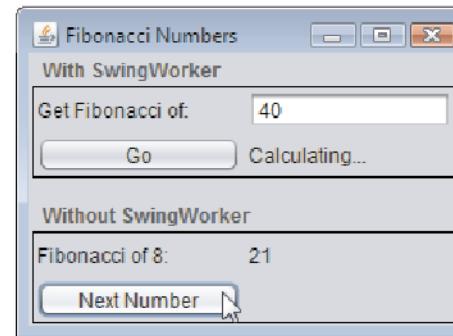


Figura 26.26 | Utilizando SwingWorker para fazer um cálculo longo exibindo os resultados em uma GUI.
(Parte 6 de 7.)



COMO PROGRAMAR

8^a edição

c) O cálculo de uma sequência Fibonacci de 40 números é finalizado.

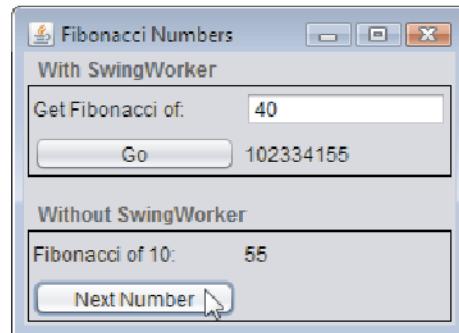


Figura 26.26 | Utilizando SwingWorker para fazer um cálculo longo exibindo os resultados em uma GUI.
(Parte 7 de 7.)



COMO PROGRAMAR

8^a edição

26.11.2 Processando resultados intermediários com SwingWorker

- ▶ A Figura 26.27 apresenta a classe **PrimeCalculator**, que estende **Swingworker** para calcular os primeiros n números primos em uma thread trabalhadora.
- ▶ Além dos métodos **doInBackground** e **done** usados no exemplo anterior, essa classe usa os métodos **Swingworker publish**, **process** e **setProgress**.
- ▶ Nesse exemplo, o método **publish** envia números primos ao método **process** à medida que eles são localizados, o método **process** exibe esses primos em um componente GUI e o método **setProgress** atualiza a propriedade de progresso.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.27: PrimeCalculator.java
2 // Calcula os n primeiros primos, exibindo-os à medida que são localizados.
3 import javax.swing.JTextArea;
4 import javax.swing.JLabel;
5 import javax.swing.JButton;
6 import javax.swing.SwingWorker;
7 import java.util.Random;
8 import java.util.List;
9 import java.util.concurrent.CancellationException;
10 import java.util.concurrent.ExecutionException;
11
12 public class PrimeCalculator extends SwingWorker< Integer, Integer >
13 {
14     private final Random generator = new Random();
15     private final JTextArea intermediateJTextArea; // exibe os primos localizados
16     private final JButton getPrimesJButton;
17     private final JButton cancelJButton;
18     private final JLabel statusJLabel; // exibe o status do cálculo
19     private final boolean[] primes; // array booleano para localizar primos
20 }
```

Figura 26.27 | Calcula os n primeiros primos, exibindo-os à medida que são localizados. (Parte I de 6.)



COMO PROGRAMAR

8^a edição

```
21 // construtor
22 public PrimeCalculator( int max, JTextArea intermediate, JLabel status,
23                         JButton getPrimes, JButton cancel )
24 {
25     intermediateJTextArea = intermediate;
26     statusJLabel = status;
27     getPrimesJButton = getPrimes;
28     cancelJButton = cancel;
29     primes = new boolean[ max ];
30
31     // inicializa todos os valores de array primos como verdadeiros
32     for ( int i = 0; i < max; i ++ )
33         primes[ i ] = true;
34 } // fim do construtor
35
36 // localiza todos os primos até o máximo utilizando o Crivo de Eratóstenes
37 public Integer doInBackground()
38 {
39     int count = 0; // o número de primos localizados
40 }
```

Figura 26.27 | Calcula os n primeiros primos, exibindo-os à medida que são localizados. (Parte 2 de 6.)



COMO PROGRAMAR

8^a edição

```
41      // iniciando no terceiro valor, circula pelo array e coloca
42      // falso como o valor de qualquer número maior que for um múltiplo
43      for ( int i = 2; i < primes.length; i++ )
44      {
45          if ( isCancelled() ) // se o cálculo tiver sido cancelado
46              return count;
47          else
48          {
49              setProgress( 100 * ( i + 1 ) / primes.length );
50
51              try
52              {
53                  Thread.sleep( generator.nextInt( 5 ) );
54              } // fim do try
55              catch ( InterruptedException ex )
56              {
57                  statusJLabel.setText( "Worker thread interrupted" );
58                  return count;
59              } // fim do catch
60
61              if ( primes[ i ] ) // i é primo
62              {
```

Figura 26.27 | Calcula os n primeiros primos, exibindo-os à medida que são localizados. (Parte 3 de 6.)



COMO PROGRAMAR

8^a edição

```
63         publish( i ); // disponibiliza para exibição na lista de primos
64         ++count;
65
66         for ( int j = i + i; j < primes.length; j += i )
67             primes[ j ] = false; // i não é primo
68     } // fim do if
69 } // fim de else
70 } // for final
71
72     return count;
73 } // fim do método doInBackground
74
75 // exibe valores publicados na lista de primos
76 protected void process( List< Integer > publishedVals )
77 {
78     for ( int i = 0; i < publishedVals.size(); i++ )
79         intermediateTextArea.append( publishedVals.get( i ) + "\n" );
80 } // fim do método process
81
```

Figura 26.27 | Calcula os n primeiros primos, exibindo-os à medida que são localizados. (Parte 4 de 6.)



COMO PROGRAMAR

8^a edição

```
82     // código para executar quando doInBackground se completa
83     protected void done()
84     {
85         getPrimesJButton.setEnabled( true ); // ativa o botão Get Primes
86         cancelJButton.setEnabled( false ); // desativa o botão Cancel
87
88         int numPrimes;
89
90         try
91         {
92             numPrimes = get(); // recupera o valor de retorno de doInBackground
93         } // fim do try
94         catch ( InterruptedException ex )
95         {
96             statusJLabel.setText( "Interrupted while waiting for results." );
97             return;
98         } // fim do catch
99         catch ( ExecutionException ex )
100        {
101            statusJLabel.setText( "Error performing computation." );
102            return;
103        } // fim do catch
```

Figura 26.27 | Calcula os n primeiros primos, exibindo-os à medida que são localizados. (Parte 5 de 6.)



COMO PROGRAMAR

8^a edição

```
104     catch ( CancellationException ex )
105     {
106         statusJLabel.setText( "Cancelled." );
107         return;
108     } // fim do catch
109
110     statusJLabel.setText( "Found " + numPrimes + " primes." );
111 } // fim do método done
112 } // fim da classe PrimeCalculator
```

Figura 26.27 | Calcula os n primeiros primos, exibindo-os à medida que são localizados. (Parte 6 de 6.)



COMO PROGRAMAR

8^a edição

- ▶ A classe `PrimeCalculator` estende `Swingworker`, com o primeiro parâmetro de tipo que indica o tipo de retorno do método `doInBackground` e o segundo que indica que o tipo de resultados intermediários passados entre os métodos `publish` e `process`.
- ▶ O método `Swingworker isCancelled` determina se o usuário clicou no botão Cancel.
- ▶ Se `isCancelled` retornar `true`, o método `doInBackground` retorna o número de primos encontrados até agora sem concluir o cálculo.
- ▶ Se o cálculo não for cancelado, a linha 49 chamará `setProgress` para atualizar a porcentagem do array que foi percorrida até agora.
- ▶ A linha 61 testa se o elemento do array `primes` no índice atual é `true` (e, portanto, primo).
- ▶ Nesse caso, a linha 63 passa o índice para o método `publish` para que ele possa ser exibido como um resultado intermediário na GUI.



COMO PROGRAMAR

8^a edição

- ▶ O método **process** executa na thread de despacho de eventos e recebe seu argumento **publishedvals** do método **publish**.
- ▶ A passagem de valores entre **publish** na thread trabalhadora e **process** na thread de despacho de eventos é assíncrona; **process** pode não ser invocado por toda chamada para **publish**.



COMO PROGRAMAR

8^a edição

```
1 // Figura 26.28: FindPrimes.java
2 // Utilizando um SwingWorker para exibir números primos e atualizar um JProgressBar
3 // enquanto os números primos estão sendo calculados.
4 import javax.swing.JFrame;
5 import javax.swing.JTextField;
6 import javax.swing.JTextArea;
7 import javax.swing.JButton;
8 import javax.swing.JProgressBar;
9 import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.ScrollPaneConstants;
13 import java.awt.BorderLayout;
14 import java.awt.GridLayout;
15 import java.awt.event.ActionListener;
16 import java.awt.event.ActionEvent;
17 import java.util.concurrent.ExecutionException;
18 import java.beans.PropertyChangeListener;
19 import java.beans.PropertyChangeEvent;
20
```

Figura 26.28 | Utilizando um SwingWorker para exibir números primos e atualizar uma JProgressBar enquanto os números primos estão sendo calculados. (Parte 1 de 7.)



COMO PROGRAMAR

8^a edição

```
21  public class FindPrimes extends JFrame
22  {
23      private final JTextField highestPrimeJTextField = new JTextField();
24      private final JButton getPrimesJButton = new JButton( "Get Primes" );
25      private final JTextArea displayPrimesJTextArea = new JTextArea();
26      private final JButton cancelJButton = new JButton( "Cancel" );
27      private final JProgressBar progressBarJProgressBar = new JProgressBar();
28      private final JLabel statusJLabel = new JLabel();
29      private PrimeCalculator calculator;
30
31      // construtor
32      public FindPrimes()
33      {
34          super( "Finding Primes with SwingWorker" );
35          setLayout( new BorderLayout() );
36
37          // inicializa o painel para obter um número do usuário
38          JPanel northJPanel = new JPanel();
39          northJPanel.add( new JLabel( "Find primes less than: " ) );
40          highestPrimeJTextField.setColumns( 5 );
41          northJPanel.add( highestPrimeJTextField );
```

Figura 26.28 | Utilizando um SwingWorker para exibir números primos e atualizar uma JProgressBar enquanto os números primos estão sendo calculados. (Parte 2 de 7.)



COMO PROGRAMAR

8^a edição

```
42     getPrimesJButton.addActionListener(
43         new ActionListener()
44     {
45         public void actionPerformed( ActionEvent e )
46     {
47         progressJProgressBar.setValue( 0 ); // redefine JProgressBar
48         displayPrimesJTextArea.setText( "" ); // limpa JTextArea
49         statusJLabel.setText( "" ); // limpa JLabel
50
51         int number; // pesquisa primos para cima até esse valor
52
53         try
54     {
55             // obtém entrada de usuário
56             number = Integer.parseInt(
57                 highestPrimeJTextField.getText() );
58         } // fim do try
59         catch ( NumberFormatException ex )
60     {
61             statusJLabel.setText( "Enter an integer." );
62             return;
63         } // fim do catch
```

Figura 26.28 | Utilizando um SwingWorker para exibir números primos e atualizar uma JProgressBar enquanto os números primos estão sendo calculados. (Parte 3 de 7.)



COMO PROGRAMAR

8^a edição

```
64
65          // constrói um novo objeto PrimeCalculator
66  calculator = new PrimeCalculator( number,
67          displayPrimesJTextArea, statusJLabel, getPrimesJButton,
68          cancelJButton );
69
70          // ouve alterações de propriedade na barra de progresso
71  calculator.addPropertyChangeListener(
72          new PropertyChangeListener()
73          {
74              public void propertyChange( PropertyChangeEvent e )
75              {
76                  // se a propriedade alterada for progress,
77                  // atualiza a barra de progresso
78                  if ( e.getPropertyName().equals( "progress" ) )
79                  {
80                      int newValue = ( Integer ) e.getNewValue();
81                      progressJProgressBar.setValue( newValue );
82                  } // fim do if
83              } // fim do método propertyChange
84          } // fim da classe interna anônima
85      ); // termina a chamada para addPropertyChangeListener
```

Figura 26.28 | Utilizando um SwingWorker para exibir números primos e atualizar uma JProgressBar enquanto os números primos estão sendo calculados. (Parte 4 de 7.)



COMO PROGRAMAR

8^a edição

```
86
87      // desativa o botão Get Primes e ativa o botão Cancel
88      getPrimesJButton.setEnabled( false );
89      cancelJButton.setEnabled( true );
90
91      calculator.execute(); // executa o objeto PrimeCalculator
92  } // fim do método actionPerformed
93 } // fim da classe interna anônima
94 ); // fim da chamada para addActionListener
95 northJPanel.add( getPrimesJButton );
96
97 // adiciona uma JList rolável para exibir os resultados do cálculo
98 displayPrimesJTextArea.setEditable( false );
99 add( new JScrollPane( displayPrimesJTextArea,
100     ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
101     ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER ) );
102
103 // inicializa um painel para exibir cancelJButton,
104 // progressJProgressBar e statusJLabel
105 JPanel southJPanel = new JPanel( new GridLayout( 1, 3, 10, 10 ) );
106 cancelJButton.setEnabled( false );
```

Figura 26.28 | Utilizando um SwingWorker para exibir números primos e atualizar uma JProgressBar enquanto os números primos estão sendo calculados. (Parte 5 de 7.)



COMO PROGRAMAR

8^a edição

```
I07     cancelJButton.addActionListener(
I08         new ActionListener()
I09         {
I10             public void actionPerformed( ActionEvent e )
I11             {
I12                 calculator.cancel( true ); // cancela o cálculo
I13             } // fim do método actionPerformed
I14         } // fim da classe interna anônima
I15     ); // fim da chamada para addActionListener
I16     southJPanel.add( cancelButton );
I17     progressJProgressBar.setStringPainted( true );
I18     southJPanel.add( progressJProgressBar );
I19     southJPanel.add( statusJLabel );
I20
I21     add( northJPanel, BorderLayout.NORTH );
I22     add( southJPanel, BorderLayout.SOUTH );
I23     setSize( 350, 300 );
I24     setVisible( true );
I25 } // fim do construtor
I26
```

Figura 26.28 | Utilizando um SwingWorker para exibir números primos e atualizar uma JProgressBar enquanto os números primos estão sendo calculados. (Parte 6 de 7.)

COMO PROGRAMAR

8ª edição

```
I27     // método main inicia a execução de programa
I28     public static void main( String[] args )
I29     {
I30         FindPrimes application = new FindPrimes();
I31         application.setDefaultCloseOperation( EXIT_ON_CLOSE );
I32     } // fim de main
I33 } // fim da classe FindPrimes
```

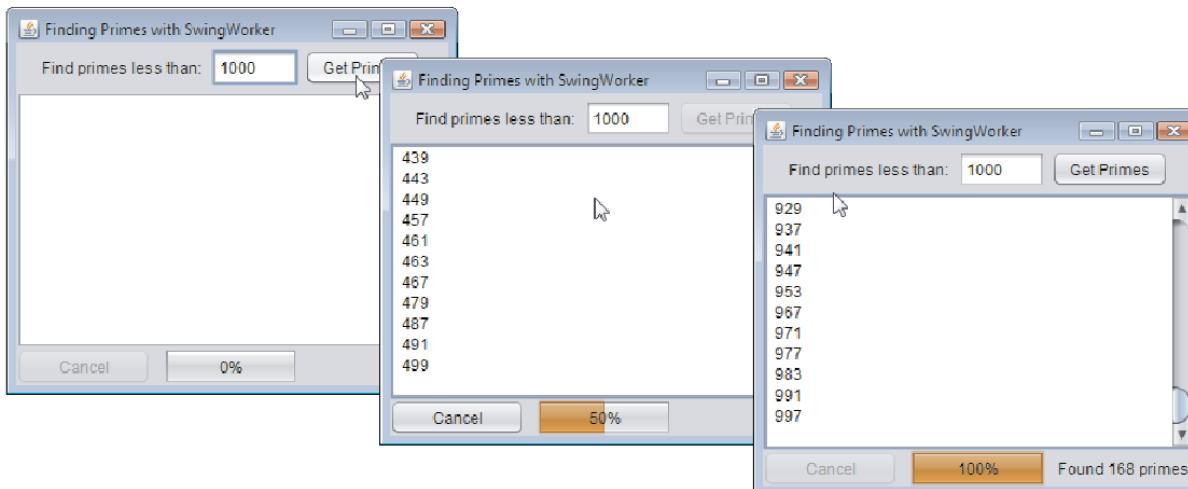


Figura 26.28 | Utilizando um SwingWorker para exibir números primos e atualizar uma JProgressBar enquanto os números primos estão sendo calculados. (Parte 7 de 7.)



COMO PROGRAMAR

8^a edição

- ▶ **PropertyChangeListener** é uma interface do pacote `java.beans` que define um único método, `propertyChange`.
- ▶ Toda vez que o método `setProgress` é invocado em um `PrimeCalculator`, o `PrimeCalculator` gera um `PropertyChangeEvent` para indicar que a propriedade de progresso mudou.
- ▶ O método `propertyChange` ouve esses eventos.
- ▶ O argumento `true` para o método `Swingworker cancel` indica que a thread que realiza a tarefa deve ser interrompida em uma tentativa de cancelar a tarefa.



COMO PROGRAMAR

8^a edição

26.12 Interfaces Callable e Future

- ▶ A interface **Callable** (do pacote `java.util.concurrent`) declara um único método chamado **call**.