# 23

# Multithreading

# OBJETIVOS

- Neste capítulo, você aprenderá:
- O que são as threads e por que elas são úteis.
- Como as threads permitem gerenciar atividades concorrentes.
- O ciclo de vida de uma thread.
- Prioridades e agendamento de threads.
- Como criar e executar `Runnable`s.
- Sincronismo de threads.
- O que são relacionamentos produtor/consumidor e como são implementados com multithreading.
- Como exibir a saída de múltiplas threads em uma GUI Swing.

# 23.1 Introdução

- **Multithreading:**
  - **Fornece múltiplas threads de execução para a aplicação.**
  - **Permite que programas realizem tarefas *concorrentemente*.**
  - **Com freqüência, exige que o programador sincronize as threads para que funcionem corretamente.**

# Dica de desempenho 23.1

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

# Dica de portabilidade 23.1

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

# 23.2 Estados de thread: Classe `Thread`

- **Estados de thread:**
  - **Estado *novo:***
    - **Uma nova thread inicia seu ciclo de vida no estado *novo*.**
    - **Permanece nesse estado até o programa iniciar a thread, colocando-a no estado *executável***
  - **Estado *executável:***
    - **Uma thread que entra nesse estado está executando sua tarefa.**
  - **Estado *em espera:***
    - **Uma thread entra nesse estado a fim de esperar que uma outra thread realize uma tarefa.**

# 23.2 Estados de thread: Classe `Thread` (*Continuação*)

- **Estados de thread:**
  - **Estado de *espera cronometrada*:**
    - **Uma thread entra nesse estado para esperar uma outra thread ou para transcorrer um determinado período de tempo.**
    - **Uma thread nesse estado retorna ao estado *executável* quando ela é sinalizada por uma outra thread ou quando o intervalo de tempo especificado expirar.**
  - **Estado *terminado:***
    - **Uma thread *executável* entra nesse estado quando completa sua tarefa.**

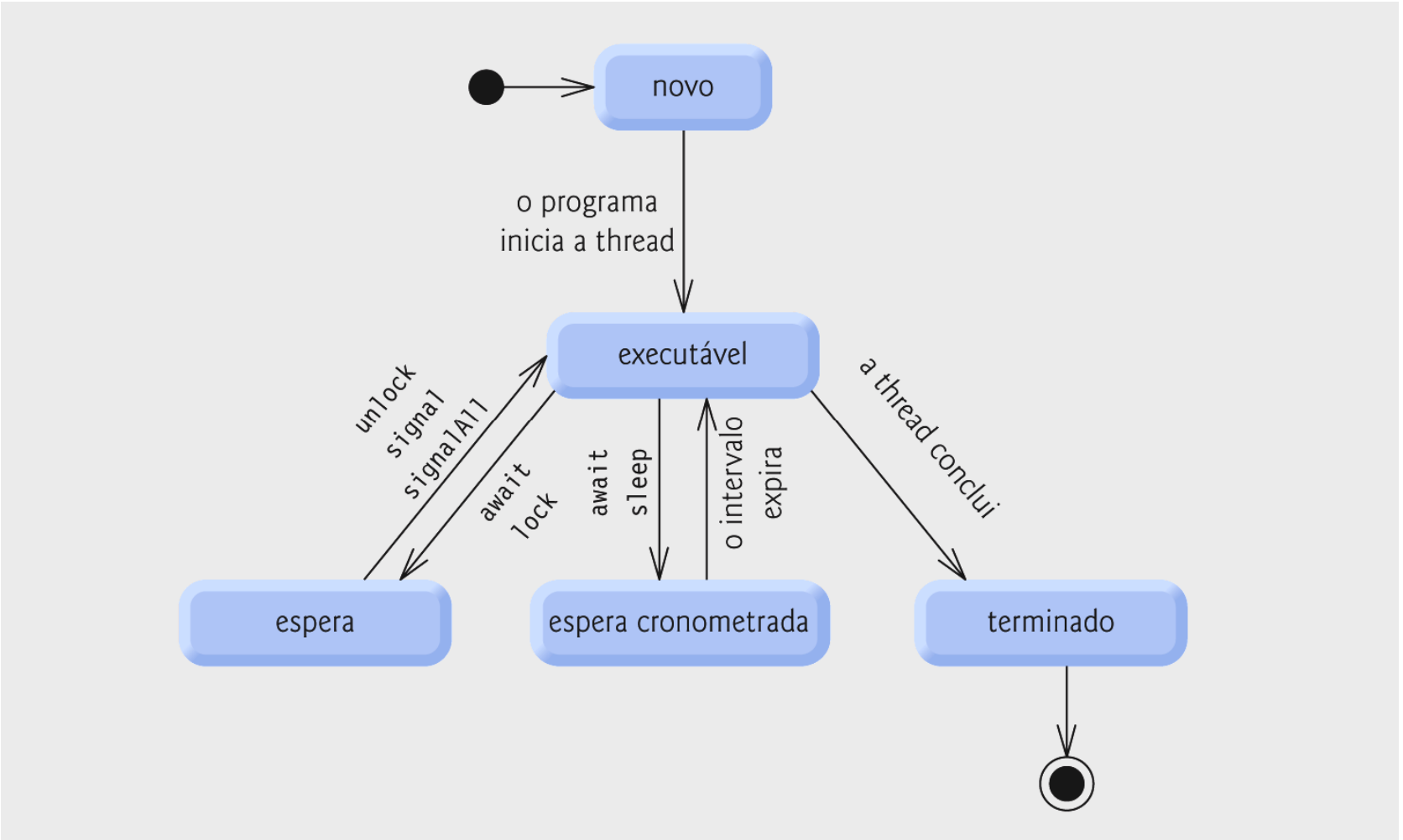


**Figura 23.1 | Diagrama de estado do ciclo de vida da thread.**

# 23.2 Estados de thread: Classe `Thread` (*Continuação*)

- **Visão do sistema operacional do estado *executável:***
  - **Estado *pronto:***
    - **Uma thread nesse estado não está esperando uma outra thread, mas está esperando que o sistema operacional atribua a thread a um processador.**
  - **Estado *em execução:***
    - **Uma thread nesse estado tem atualmente um processador e está executando.**
  - **Uma thread no estado *em execução* freqüentemente utiliza uma pequena quantidade de tempo de processador chamada fração de tempo, ou *quantum*, antes de migrar de volta para o estado *pronto*.**

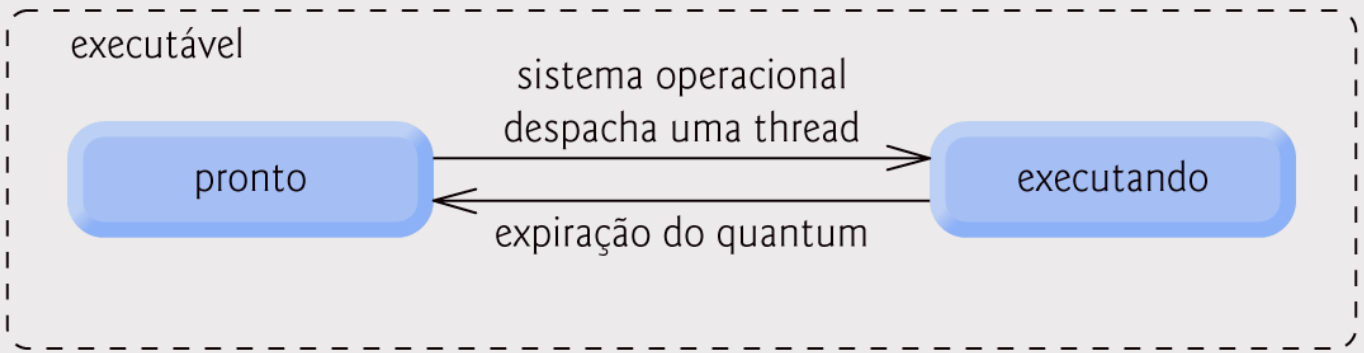


**Figura 23.2 | Visualização interna do sistema operacional do estado executável do Java.**

# 23.3 Prioridades de thread e agendamento de thread

- **Prioridades:**
  - **Cada thread Java tem uma prioridade.**
  - **As prioridades do Java estão no intervalo entre `MIN_PRIORITY` (uma constante de 1) e `MAX_PRIORITY` (uma constante de 10).**
  - **As threads com uma prioridade mais alta são mais importantes e terão um processador alocado antes das threads com uma prioridade mais baixa.**
  - **A prioridade-padrão é `NORM_PRIORITY` (uma constante de 5).**

# 23.3 Prioridades de thread e agendamento de thread (*Cont.*)

- **Agendador de thread:**
  - **Determina qual thread é executada em seguida.**
  - **Uma implementação simples executa threads com a mesma prioridade no estilo *rodízio*.**
  - **Threads de prioridade mais alta podem fazer preempção da thread atualmente *em execução*.**
  - **Em alguns casos, as threads de prioridade alta podem adiar indefinidamente threads de prioridade mais baixa — o que também é conhecido como *inanição*.**

# Dica de portabilidade 23.2

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

# Dica de portabilidade 23.3

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

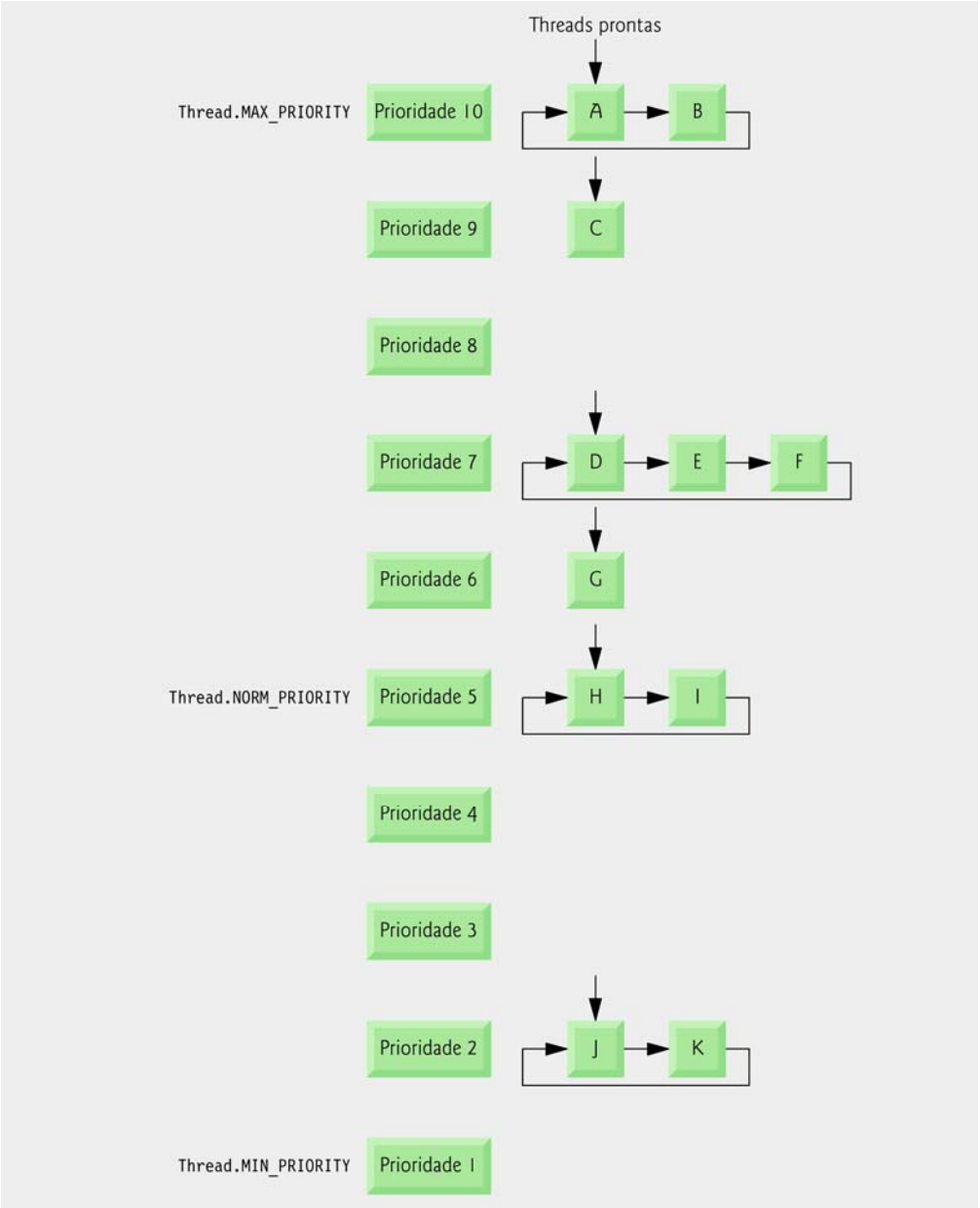


**Figura 23.3 | Agendamento de prioridade de threads.**

# 23.4 Criando e executando threads

- **A interface `Runnable`:**
  - **Meio preferido de criar um aplicativo com multithreads.**
  - **Declara o método `run`.**
  - **Executado por um objeto que implementa a interface `Executor`.**
- **Interface `Executor`:**
  - **Declara o método `execute`.**
  - **Cria e gerencia um grupo de threads chamado *pool de threads*.**

# 23.4 Criando e executando threads (*Continuação*)

- **Interface `ExecutorService`:**
  - **É uma subinterface de `Executor` que declara outros métodos para gerenciar o ciclo de vida de um `Executor`.**
  - **Pode ser criada utilizando os métodos `static` da classe `Executors`.**
  - **O método `shutdown` finaliza as threads quando as tarefas são concluídas.**
- **Classe `Executors`:**
  - **O método `newFixedThreadPool` cria um pool que consiste em um número fixo de threads.**
  - **O método `newCachedThreadPool` cria um pool que cria novas threads conforme necessário.**

# Resumo

**PrintTask.java**

```
1  // Fig. 23.4: PrintTask.java
2  // Classe PrintTask dorme por um tempo aleatório de 0 a 5 segundos
3  import java.util.Random;
4
5  class PrintTask implements Runnable
6  {
7     private int sleepTime; // tempo de adormecimento aleatório para a thread
8     private String threadName; // nome da thread
9     private static Random generator = new Random();
10
11    // atribui nome à thread
12    public PrintTask( String name )
13    {
14       threadName = name; // configura nome da thread
15
16       // seleciona tempo de adormecimento aleatório entre 0 e 5 segundos
17       sleepTime = generator.nextInt( 5000 );
18    } // fim do construtor PrintTask
19
```

Implementa `runnable` para criar uma thread separada

# Resumo

**PrintTask.java**

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

Declara o método `run` para atender a interface

# Resumo

**RunnableTester**

```
1  // Fig. 23.5: RunnableTester.java
2  // Impressão de múltiplas threads em diferentes intervalos.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5
6  public class RunnableTester
7  {
8     public static void main( String[] args )
9     {
10       // cria e nomeia cada executável
11       PrintTask task1 = new PrintTask( "thread1" );
12       PrintTask task2 = new PrintTask( "thread2" );
13       PrintTask task3 = new PrintTask( "thread3" );
14
15       System.out.println( "Starting threads" );
16
17       // cria ExecutorService para gerenciar threads
18       ExecutorService threadExecutor = Executors.newFixedThreadPool( 3 );
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
24
25       threadExecutor.shutdown(); // encerra as threads
26
```

Cria três `PrintTask`s; cada uma executará em uma thread separada

Cria um pool de threads fixas para executar e gerenciar threads

Executa cada tarefa; esse método atribuirá uma thread a `runnable`

Desativa o pool de threads quando os `runnable`s completarem suas tarefas

```
27       System.out.println( "Threads started, main ends\n" );
28    } // fim do main
29 } // fim da classe RunnableTester
```

```
Starting threads
Threads started, main ends

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

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

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

# Resumo

**RunnableTester .java**

(2 de 2)

# 23.5 Sincronismo de thread

- **Sincronismo de threads:**
  - **Fornecido ao programador com exclusão mútua.**
    - **Acesso exclusivo a um objeto compartilhado.**
  - **Implementado no Java utilizando bloqueios.**
- **Interface `Lock`:**
  - **O método `lock` obtém o bloqueio, impondo a exclusão mútua.**
  - **O método `unlock` libera o bloqueio.**
  - **A classe `ReentrantLock` implementa a interface `Lock`.**

# Dica de desempenho 23.2

**Utilizar um `Lock` com uma diretiva relativamente justa evita o adiamento indefinido, mas também pode reduzir significativamente a eficiência geral de um programa. Por causa da grande diminuição de desempenho, os bloqueios imparciais só são necessários em circunstâncias extremas.**

# 23.5 Sincronismo de thread (*Continuação*)

- **Variáveis de condição:**
  - **Se uma thread que mantém o bloqueio não puder continuar a sua tarefa até uma condição ser satisfeita, a thread pode esperar uma *variável de condição*.**
  - **Criadas chamando `newCondition` do método `Lock`.**
  - **Representadas por um objeto que implementa a interface `Condition`.**
- **Interface `Condition`:**
  - **Declara os métodos: `await`, para fazer uma thread esperar; `signal`, para acordar uma thread em espera; e `signalAll`, para acordar todas as threads em espera.**

# Erro comum de programação 23.1

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

# Dica de prevenção de erro 23.1

**Quando múltiplas threads manipulam um objeto compartilhado utilizando bloqueios, assegure de que, se uma thread chamar o método `await` para entrar no estado de espera por uma variável de condição, uma thread separada por fim chamará o método `Condition signal` para fazer a transição da thread em espera pela variável de condição de volta para o estado *executável*. (*Continua*...)**

# Dica de prevenção de erro 23.1

**Se múltiplas threads podem estar esperando a variável de condição, uma thread separada pode chamar o método `Condition signalAll` como uma salvaguarda para assegurar que todas as threads na espera tenham outra oportunidade de realizar suas tarefas. Se isso não for feito, o adiamento indefinido ou impasse poderia ocorrer.**

# Observação de engenharia de software 23.1

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

# Dica de desempenho 23.3

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

# Erro comum de programação 23.2

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

# 23.6 Relacionamento entre produtor e consumidor sem sincronismo

- **Relacionamento produtor/consumidor:**
  - **O produtor gera dados e os armazena na memória compartilhada.**
  - **O consumidor lê os dados da memória compartilhada.**
  - **A memória compartilhada é chamada *buffer*.**

# Resumo

**Bbuffer.java**

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

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

# Resumo

**Producer.java**

```
1  // Fig. 23.7: Producer.java
2  // O método run do Producer armazena os valores de 1 a 10 no buffer.
3  import java.util.Random;
4
5  public class Producer implements Runnable
6  {
7     private static Random generator = new Random();
8     private Buffer sharedLocation; // referência a obje
9
10    // construtor
11    public Producer( Buffer shared )
12    {
13        sharedLocation = shared;
14    } // fim do construtor Producer
15
16    // armazena valores de 1 a 10 em sharedLocation
17    public void run()
18    {
19       int sum = 0;
20
```

Implementa a interface `runnable` de modo que o produtor possa ser executado em uma thread separada

Declara o método `run` para satisfazer a interface

# Resumo

**Producer.java**

(2 de 2)

```
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 ) ); // thread sleep
26             sharedLocation.set( count ); // configura valor no buffer
27             sum += count; // incrementa soma de valores
28             System.out.printf( "\t%2d\n", sum );
29          } // fim do try
30          // se a thread adormecida é interrompida, imprime rastreamento de pilha
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // fim do catch
35       } // fim do for
36
37       System.out.printf( "\n%s\n%s\n", "Producer done producing.",
38          "Terminating Producer." );
39    } // fim do método run
40 } // fim da classe Producer
```

Dorme por até 3 segundos

# Resumo

Consumer.java

```
1  // Fig. 23.8: Consumer.java
2  // O método run de Consumer itera dez vezes lendo um valor do buffer.
3  import java.util.Random;
4
5  public class Consumer implements Runnable
6  {
7     private static Random generator = new Random();
8     private Buffer sharedLocation; // referência a objet
9
10    // construtor
11    public Consumer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // fim do construtor Consumer
15
16    // lê o valor do sharedLocation quatro vezes e soma
17    public void run()
18    {
19       int sum = 0;
20
```

Implementa a interface `runnable` de modo que o produtor possa ser executado em uma thread separada

Declara o método `run` para satisfazer a interface

# Resumo

**Consumer.java**

(2 de 2)

```
21       for ( int count = 1; count <= 10; count++ )
22       {
23          // dorme de 0 a 3 segundos, lê o valor do buffer e adiciona a soma
24          try
25          {
26             Thread.sleep( generator.nextInt( 3000 ) );
27             sum += sharedLocation.get();
28             System.out.printf( "\t\t\t%2d\n", sum );
29          } // fim do try
30          // se a thread adormecida é interrompida, imp
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // fim do catch
35       } // fim do for
36
37       System.out.printf( "\n%s %d.\n%s\n",
38          "Consumer read values totaling", sum, "Terminating Consumer." );
39    } // fim do método run
40 } // fim da classe Consumer
```

Dorme por até 3 segundos

# Resumo

**UnsynchronizedBuffer.java**

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

Variável compartilhada para armazenar dados

Configura o valor do buffer

Lê o valor do buffer

# Resumo

**SharedBufferTest.java**

(1 de 4)

```
1  // Fig 23.10: SharedBufferTest.java
2  // Aplicativo mostra duas threads que manipulam um buffer não-sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // cria UnsynchronizedBuffer para armazenar ints
14       Buffer sharedLocation = new UnsynchronizedBuffer();
15
```

Cria um `UnsynchronizedBuffer` compartilhado para que o produtor e o consumidor o utilizem

# Resumo

**SharedBufferTest.java**

(2 de 4)

```
16       System.out.println( "Action\t\tValue\tProduced\tConsumed" );
17       System.out.println( "------\t\t-----\t--------\t--------\n" );
18
19       // tenta iniciar as threads produtora e consumidora fornecendo acesso a cada uma
20       // para sharedLocation
21       try
22       {
23          application.execute( new Producer( sharedLocation ) );
24          application.execute( new Consumer( sharedLocation ) );
25       } // fim do try
26       catch ( Exception exception )
27       {
28          exception.printStackTrace();
29       } // fim do catch
30
31       application.shutdown(); // termina aplicativo quando as threads terminam
32    } // fim do main
33 } // fim da classe SharedBufferTest
```

Passa o buffer compartilhado tanto para o produtor como para o consumidor

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

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

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

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

# Resumo

**SharedBufferTest.java**

(3 de 4)

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

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

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

Producer done producing.
Terminating Producer.
```

# Resumo

**SharedBufferTest.java**

(4 de 4)

# 23.7 Relacionamento entre produtor e consumidor com sincronismo

- **Relacionamento produtor/consumidor:**
  - **Este exemplo utiliza `Locks` e `Conditions` para implementar a sincronismo.**

# Resumo

```
1  // Fig. 23.11: SynchronizedBuffer.java
2  // SynchronizedBuffer sincroniza acesso a um único inteiro compartilhado.
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5  import java.util.concurrent.locks.Condition;
6
7  public class SynchronizedBuffer implements Buffer
8  {
9     // Bloqueio para controlar sincronização com esse buffer
10    private Lock accessLock = new ReentrantLock();
11
12    // condições para controlar leitura e gravação
13    private Condition canWrite = accessLock.newCondition();
14    private Condition canRead = accessLock.newCondition();
15
16    private int buffer = -1; // compartilhado pelas thre
17    private boolean occupied = false; // se o buffer est
18
19    // coloca o valor int no buffer
20    public void set( int value )
21    {
22       accessLock.lock(); // bloqueia esse objeto
23
```

izedBuffer

(1 de 5)

Cria `ReentrantLock` para exclusão mútua

Cria duas variáveis de `Condition`; uma para gravação e outra para leitura

Buffer compartilhado por produtor e consumidor

Tenta obter o bloqueio antes de configurar o valor dos dados compartilhados

# Resumo

**SynchronizedBuffer.java**

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

Produtor espera até que o buffer esteja vazio

# Resumo

```
41          displayState( "Producer writes " + buffer );
42
43          // sinaliza a thread que está esperando para ler a partir do buffer
44          canRead.signal();
45       } // fim do try
46       catch ( InterruptedException exception )
47       {
48          exception.printStackTrace();
49       } // fim do catch
50       finally
51       {
52          accessLock.unlock(); // desbloqueia esse objeto
53       } // fim do finally
54    } // fim do método set
55
56    // retorna valor do buffer
57    public int get()
58    {
59       int readValue = 0; // inicializa de valor lido a partir do buffer
60       accessLock.lock(); // bloqueia esse objeto
61
```

Sinaliza ao consumidor que ele pode ler um valor

Libera o bloqueio sobre os dados compartilhados

Adquire o bloqueio antes de ler um valor

**SynchronizedBuffer.java**

(3 de 5)

# Resumo

**SynchronizedBuffer.java**

```
62       // envia informações de thread e de buffer para a saída, então espera
63       try
64       {
65          // enquanto os dados não são lidos, coloca thread em estado de espera
66          while ( !occupied )
67          {
68             System.out.println( "Consumer tries to read." );
69             displayState( "Buffer empty. Consumer waits." );
70             canRead.await(); // espera até o buffer tor
71          } // fim do while
72
73          // indica que a produtora pode armazenar outro valor
74          // porque a consumidora acabou de recuperar o valor do buffer
75          occupied = false;
76
77          readValue = buffer; // recupera o valor do buffer
78          displayState( "Consumer reads " + readValue );
79
```

O consumidor espera até que o buffer contenha os dados a ler

```
80          // sinaliza a thread que está esperando o buffer tornar-se vazio
81          canWrite.signal(); // sinaliza ao produtor que ele pode gravar no buffer
82       } // fim do try
83       // se a thread na espera tiver sido interrompida, imprime
84       catch ( InterruptedException exception )
85       {
86          exception.printStackTrace();
87       } // fim do catch
88       finally
89       {
90          accessLock.unlock(); // desbloqueia esse objeto
91       } // fim do finally
92
93       return readValue;
94    } // fim do método get
95
96    // exibe a operação atual e o estado de buffer
97    public void displayState( String operation )
98    {
99       System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
100         occupied );
101   } // fim do método displayState
102} // fim da classe SynchronizedBuffer
```

Sinaliza ao produtor que ele pode gravar no buffer

Libera o bloqueio sobre os dados compartilhados

mo

**SynchronizedBuffer.java**

(5 de 5)

# Erro comum de programação 23.3

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

# Observação de engenharia de software 23.2

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

# Erro comum de programação 23.4

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

# Resumo

**SharedBufferTest2.java**

(1 de 4)

```
1  // Fig 23.12: SharedBufferTest2.java
2  // Aplicativo mostra duas threads que manipulam um buffer sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // cria SynchronizedBuffer para armazenar ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
```

Cria o `SynchronizedBuffer` a ser compartilhado entre produtor e consumidor

# Resumo

**SharedBufferTest2**

```
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t--------" );
18
19       try // tenta iniciar a produtora e a consumidora
20       {
21          application.execute( new Producer( sharedLocation ) );
22          application.execute( new Consumer( sharedLocation ) );
23       } // fim do try
24       catch ( Exception exception )
25       {
26          exception.printStackTrace();
27       } // fim do catch
28
29       application.shutdown();
30    } // fim do main
31 } // fim da classe SharedBufferTest2
```

Executa o produtor e o consumidor em threads separadas

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

Producer writes 1                  1               true

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

Consumer reads 1                   1               false

Producer writes 2                  2               true

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

Consumer reads 2                   2               false

Producer writes 3                  3               true

Consumer reads 3                   3               false

Producer writes 4                  4               true

Consumer reads 4                   4               false

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

Producer writes 5                  5               true

Consumer reads 5                   5               false

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

# Resumo

**SharedBufferTest2.java**

(3 de 4)

```
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

Producer writes 10                  10              true


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


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

# Resumo

**SharedBufferTest2.java**

(4 de 4)