Java – Aula 07

Multithreading

17/10/2012

Celso Olivete Júnior

olivete@fct.unesp.br

Aula passada:

- Banco de dados – JTable

Na aula de hoje:

- Introdução
- Estados de thread: Classe Thread
- Prioridades de thread e agendamento de thread
- Criando e executando threads
- Sincronização de thread
- Relacionamento entre produtor e consumidor sem sincronização
- Relacionamento entre produtor e consumidor com sincronização
- Relacionamento de produtor/consumidor: Buffer circular
- Relacionamento de produtor/consumidor: ArrayBlockingQueue
- Multithreading com GUI
- Conclusão

Introdução

- Mundo real
  - funciona concorrentemente:
    - atividades podem ser executadas paralelamente
    - Ex: uma pessoa pode estar: respirando, falando, andando...
- Computadores também operam concorrentemente. Ex: compilando um programa, imprimindo, gravando...
  - Multithreading:
    - Fornece múltiplas threads de execução para a aplicação.
    - Permite que programas realizem tarefas concorrentemente.
    - Com frequência, exige que o programador sincronize as threads para que funcionem corretamente.

Multithreading

- 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 possa operar de modo mais eficiente.

Multithreading

- 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.

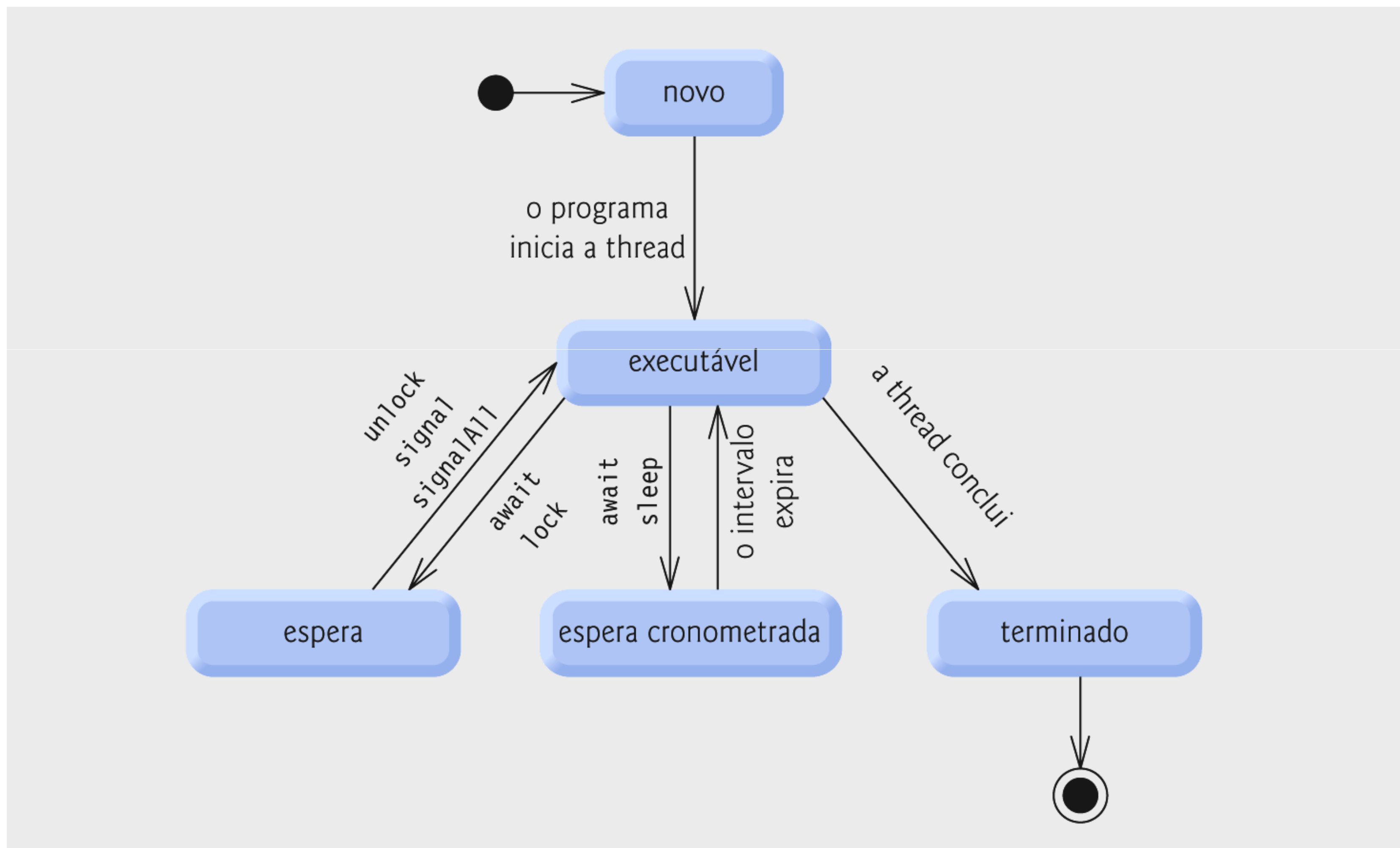
Estados de thread: Classe Thread

- 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.

Estados de thread: Classe 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. Ex: thread para salvar arquivo automaticamente
- Estado terminado:
  - Uma thread executável entra nesse estado quando completa sua tarefa.

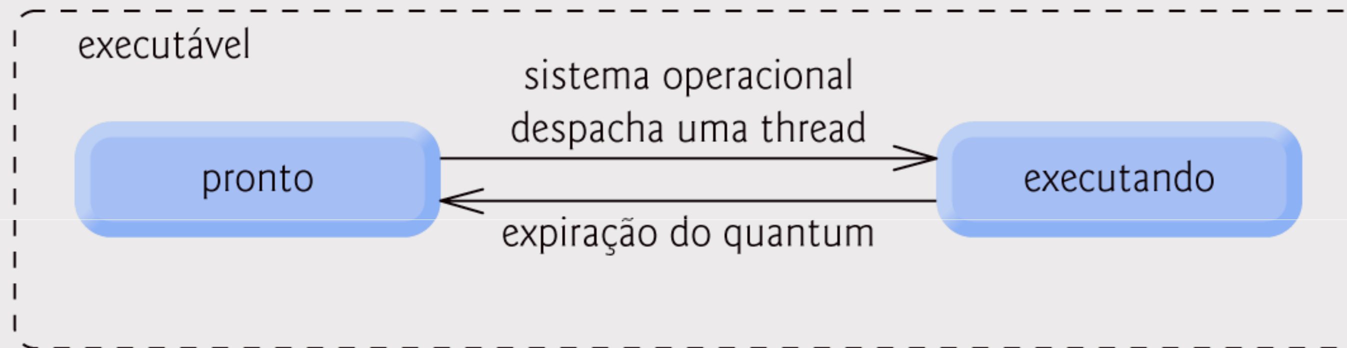
Estados de thread: Classe Thread



Estados de thread: Classe Thread

- 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 frequentemente utiliza uma pequena quantidade de tempo de processador chamada fração de tempo, ou quantum, antes de migrar de volta para o estado pronto.

Estados de thread: Classe Thread



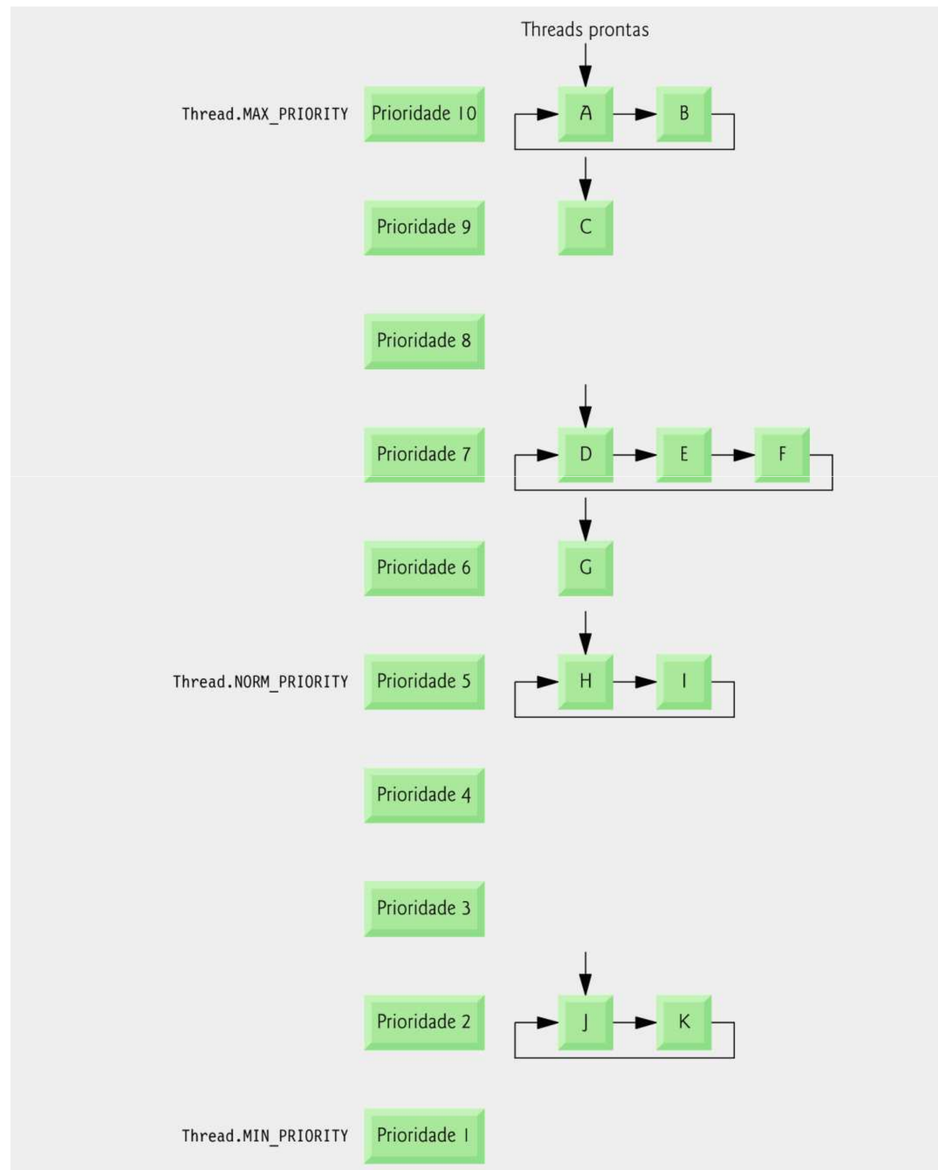
Prioridades de thread e agendamento de thread

- 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).

Prioridades de thread e agendamento de thread

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

Agendamento de prioridade de thread



Criando e executando threads

- Implementar a interface **Runnable** (pacote `java.lang`):
  - Meio preferido de criar um aplicativo com multithreads.
  - Declara um único método **run**.
  - Runnable é executado por um objeto que implementa a interface **Executor** (pacote `java.util.concurrent`).
- Interface **Executor**:
  - Declara um único método - **execute**.
  - Cria e gerencia um grupo de threads chamado **pool de threads**.
    - essas threads executam os objetos **Runnable** passados para o método **execute**
    - **Executor** atribui cada **Runnable** a cada uma das threads disponíveis no **pool de threads**

Criando e executando threads

- Interface **ExecutorService** (pacote `java.util.concurrent`):
  - 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.

Classe PrintTask

```
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 a interface **Runnable**

Todo objeto do tipo **PrintTask** pode executar **concorrentemente sem sincronismo**

Classe PrintTask

```
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
```

Método de **Runnable**
Responsável por iniciar a execução

Mostra o nome da **Thread** e o tempo que de adormecimento

Thread entra em ***espera sincronizada.***
Thread perde o processador, outra thread entra em execução

Quando terminar o tempo de adormecimento, thread volta para o estado de execução

Classe 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 trabalhadoras
26
```

Cria três objetos (threads) do tipo **PrintTask**

Cria três ***threads***

Cria um pool para especificamente três ***threads***

Coloca as threads em execução ***(Runnable)***

Finaliza as threads - após concluir execução de ***Runnable***

```
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
```

Thread principal termina antes que qualquer outra thread gere alguma saída

Threads no estado de espera sincronizada

Threads deixam o estado de espera após terminar o tempo de adormecimento

```
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
```

Exemplo de agendamento de rodízio

Sincronização de thread

- Threads manipulam objeto compartilhado na memória – **necessidade** de sincronização de acesso ao objeto
- Sincronização de threads:
  - Fornecido ao programador com exclusão mútua (objeto acessado apenas por uma thread).
  - Acesso exclusivo a um objeto compartilhado. Outras threads que desejarem acessar esse objeto terão que esperar
  - Implementado no Java com o uso de **bloqueios**.
- Interface **Lock** (pacote `java.util.concurrent.locks`):
  - O método **lock** obtém o bloqueio, impondo a exclusão mútua.
  - O método **unlock** libera o bloqueio (thread de prioridade mais alta pode acessá-lo) .
  - A classe **ReentrantLock** implementa a interface **Lock**.

Sincronização de thread

- 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 em uma variável de condição → dispensa a thread da disputa do processador
  - 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.

Relacionamento entre produtor e consumidor sem sincronização

- O produtor gera dados e os armazena na memória compartilhada.
- O consumidor lê os dados da memória compartilhada.
- A memória compartilhada é chamada buffer.
- Exemplo: spooling de impressão
  - um processador de texto (produtor) faz um spool de dados para um buffer (arquivo) e esses dados são consumidos pela impressora (consumidor) ➔ **forma sincronizada**
  - os dados vão sendo enviados pelo produtor na medida que o consumidor já leu os dados enviados anteriormente
    - caso contrário, produtor entra no estado **await** – espera consumidor consumir dados enviados anteriormente para não ocorrer sobreposição de dados
    - quando a parte consumidora terminar de processar os dados, deve chamar **signal**

Relacionamento entre produtor e consumidor sem sincronização

- Exemplo:
  - uma thread produtora grava os números de 1 a 10 (soma = 55) em um **buffer compartilhado**
  - thread consumidor lê os números do buffer compartilhado e os exibe
  - saída do programa: mostra os valores que a produtora grava (produz) no buffer e os valores que a consumidora lê (consume) a partir do buffer
  - threads dormem em intervalos aleatórios de até 3 segundos entre execução de suas tarefas (logo não se sabe quando a produtora vai gravar um novo número e quando a consumidora vai ler um novo número)
    - cada valor produzido (produtor) deve ser lido exatamente uma vez pela consumidora (consumidor)
    - – PROBLEMA: THREADS NÃO SINCRONIZADAS
      - dados podem ser perdidos se produtora tentar gravar novo número no buffer antes da consumidora ter lido o número gravado anteriormente
      - números podem ser duplicados incorretamente se a consumidora consumir números outra vez sem a produtora produzir o próximo número

Relacionamento entre produtor e consumidor sem sincronização

- Programa consiste em:
  - interface Buffer: declara os métodos **set** e **get** **(valor)** que serão utilizados, respectivamente, pela produtora e consumidora
  - classes
    - Producer (produtora)
    - Consumer (consumidora)
    - UnSynchronizedBuffer (buffer não sincronizado)
    - SharedBufferTest (testar buffer compartilhado)

Interface Buffer

```
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
```

Classe Producer

```
1  // Fig. 23.7: Producer.java
2  // O método run do Producer armazena os valores de 1 a 10 no buffer.
3  import java.util.Random;
4
5  public class Producer implements Runnable
6  {
7     private static Random generator = new Random();
8     private Buffer sharedLocation; // referência a objeto compartilhado
9
10    // construtor
11    public Producer( Buffer shared )
12    {
13        sharedLocation = shared;
14    } // fim do construtor Producer
15
16    // armazena 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, concorrentemente com **Consumer**

Inicializa o **Buffer**

Declara o método `run` para satisfazer a interface

Classe Producer

```
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

Espera sincronizada e então coloca o valor no Buffer

Concluiu a produção de dados e está terminando

Classe Consumer

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

Implementa a interface `runnable` de modo que o produtor possa ser executado em uma thread separada, concorrentemente com **Producer**

Declara o método `run` para satisfazer a interface

Classe Consumer

Dorme por até 3 segundos

Espera sincronizada e então lê o valor do Buffer

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

Classe UnSynchronizedBuffer

```
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. Caso **consumer** tenta ler um valor antes de **producer** armazenar o número

Configura o valor do buffer

Lê o valor do buffer

Classe SharedBufferTest

```
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

Classe SharedBufferTest

```
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 (mesmo) buffer compartilhado tanto para o produtor como para o consumidor. Chamam implicitamente o método **run** de cada **Runnable**

Objetivo: thread **producer** deve produzir um número para só então a **consumer** ler o número (**uma única vez!**)

```
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.
```

Producer gravou três números (total igual a 6)

Consumer leu o número 3; números 1 e 2 foram perdidos

Saída (consumer) totalizou 77 quando o correto seria 55

```
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.
```

Saída (consumer) totalizou 19 quando o correto seria 55.
Logo o acesso a um objeto compartilhado por threads concorrentes deve ser cuidadosamente controlado

Relacionamento entre produtor e consumidor com sincronização

- Este exemplo utiliza **Locks** e **Conditions (await** e **signal)** para implementar a sincronização.
  - dessa forma, quando uma thread adquire o bloqueio (**await**), nenhuma outra thread pode adquirir esse mesmo bloqueio até a thread original liberá-lo

Relacionamento entre produtor e consumidor com sincronização

- a thread consumidora consumirá exatamente o total de elementos produzidos
  - a thread consumidora consome um valor somente após a produtora ter produzido um valor

Classes **Producer** e **Consumer** são mantidas, bem como a interface **Buffer**. Criada apenas a classe **SynchronizedBuffer**

Classe SynchronizedBuffer

```
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 threads producer e consumer
17    private boolean occupied = false; // se o buffer estiver ocupado
18
19    // coloca o valor int no buffer
20    public void set( int value )
21    {
22       accessLock.lock(); // bloqueia esse objeto
23
```

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

Classe SynchronizedBuffer

```
24         // envia informações de thread e de buffer para a saída, então espera
25         try
26         {
27            // enquanto o buffer não estiver vazio, coloca thread no estado de espera
28            while ( occupied )
29            {
30               System.out.println( "Producer tries to write." );
31               displayState( "Buffer full. Producer waits." );
32               canWrite.await(); // espera até que o buffer esteja vazio
33            } // 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

Após Consumidor ler o buffer, Produtor grava um novo valor

Indica que existe um valor para ser lido no buffer – buffer ocupado

Classe SynchronizedBuffer

```
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

Classe SynchronizedBuffer

```
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

Classe SynchronizedBuffer

```
80             // sinaliza a thread que está esperando o buffer tornar-se vazio
81             canWrite.signal();
82          } // fim do try
83          // se a thread na espera tiver sido interrompida, imprime o ra
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

Classe SharedBufferTest

```
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

Classe SharedBufferTest

```
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
```

Problema: não apresenta ótimo desempenho. **Tempo perdido em espera**

```
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.
```

Problema: não apresenta ótimo desempenho.
Tempo perdido em espera.
Se consumidora ainda não leu o valor, produtora tem que aguardar para armazenar novo valor; Se produtora ainda não armazenou novo valor, consumidora tem que aguardar pra ler.

Relacionamento de produtor/ consumidor: Buffer circular

- Buffer circular:
  - Fornece espaço extra em buffer no qual o produtor pode colocar valores e o consumidor pode ler valores
    - buffer funciona como um array; produtor e consumidor funcionam bem desde o início do array
    - Ex: fila de impressão

Objetivo: tornar execução mais rápida, evitar retardos...

Classe CircularBuffer

```
1  // Fig. 23.13: CircularBuffer.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 CircularBuffer 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, -1, -1 };
17
18    private int occupiedBuffers = 0; // conta número de buffers utilizados
19    private int writeIndex = 0; // índice para escrever o próximo valor
20    private int readIndex = 0; // índice para ler o próximo valor
21
22    // coloca o valor no buffer
23    public void set( int value )
24    {
25       accessLock.lock(); // bloqueia esse objeto
26
```

Bloqueia para impor exclusão mútua

Variáveis de condição para controlar a gravação e leitura

Buffer circular; fornece três espaços para dados

Se for = 3; producer espera.

Se for = 0; consumer espera

Obtém o bloqueio antes de gravar dados no buffer circular

Consumidora só consome quando o array não estiver vazio; produtora só produz quando o array não estiver cheio

Classe CircularBuffer

```
27        // envia informações de thread e de buffer para a saída, então espera
28        try
29        {
30           // enquanto não houver posições vazias, põe o thread no estado de espera
31           while ( occupiedBuffers == buffer.length )
32           {
33              System.out.printf( "All buffers full. Producer waits.\n" );
34              canWrite.await();//espera até um elemento buffer ser liberado
35           } // fim do while
36
37           buffer[ writeIndex ] = value; // configura novo valor de buffer
38
39           // atualiza índice de gravação circular
40           writeIndex = ( writeIndex + 1 ) % buffer.length;
41
42           occupiedBuffers++; // mais um elemento buffer está cheio
43           displayState( "Producer writes " + buffer[ writeIndex ] );
44           canRead.signal();// sinaliza threads que estão esperando para ler o buffer
45        } // fim do try
46        catch ( InterruptedException exception )
47        {
48           exception.printStackTrace();
49        } // fim do catch
50        finally
51        {
52           accessLock.unlock(); // desbloqueia esse objeto
53        } // fim do finally
54     } // fim do método set
55
```

Espera até um espaço de buffer estar vazio

Atualiza o índice; essa instrução impõe a circularidade do buffer

Sinaliza a thread em espera de que agora ela pode ler dados no buffer

Libera o bloqueio

Classe CircularBuffer

```
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
62       // espera até que o buffer tenha dados, então lê o valor
63       try
64       {
65          // enquanto os dados não são lidos, coloca thread em estado de espera
66          while ( occupiedBuffers == 0 )
67          {
68             System.out.printf( "All buffers empty. Consumer waits.\n" );
69             canRead.await(); // espera até que um elemento buffer seja preenchido
70          } // fim do while
71
72          readValue = buffer[ readIndex ]; // lê valor do buffer
73
74          // atualiza índice de leitura circular
75          readIndex = ( readIndex + 1 ) % buffer.length;
76
```

Bloqueia o objeto antes de tentar ler um valor

Espera um valor a ser gravado no buffer

Atualiza o índice de leitura; essa instrução impõe a circularidade do buffer

Classe CircularBuffer

```
77          occupiedBuffers--; // mais um elemento buffer está vazio
78          displayState( "Consumer reads " + readValue );
79          canWrite.signal(); // sinaliza threads que estão esperando para gravar no buffer
80       } // fim do try
81       // se a thread na espera tiver sido interrompida, imprime
82       catch ( InterruptedException exception )
83       {
84          exception.printStackTrace();
85       } // fim do catch
86       finally
87       {
88          accessLock.unlock(); // desbloqueia esse objeto
89       } // fim do finally
90
91       return readValue;
92    } // fim do método get
93
94    // exibe operação atual e o estado do buffer
95    public void displayState( String operation )
96    {
97       // gera saída de operação e número de buffers ocupados
98       System.out.printf( "%s%s%d)\n%s", operation,
99          " (buffers occupied: ", occupiedBuffers, "buffers:  " );
100
101      for ( int value : buffer )
102         System.out.printf( " %2d  ", value ); // gera a saída dos valores no buffer
103
```

Sinaliza threads que estão esperando para gravar no buffer

Libera o bloqueio

Exibe o conteúdo do buffer

Classe CircularBuffer

```
104        System.out.print( "\n          " );
105        for ( int i = 0; i < buffer.length; i++ )
106           System.out.print( "---- " );
107
108        System.out.print( "\n          " );
109        for ( int i = 0; i < buffer.length; i++ )
110        {
111           if ( i == writeIndex && i == readIndex )
112              System.out.print( " WR" ); // índice de gravação e de leitura
113           else if ( i == writeIndex )
114              System.out.print( " W   " ); // só índice de gravação
115           else if ( i == readIndex )
116              System.out.print( "  R  " ); // só índice de leitura
117           else
118              System.out.print( "     " ); // nenhum dos índices
119        } // fim do for
120
121        System.out.println( "\n" );
122     } // fim do método displayState
123} // fim da classe CircularBuffer
```

Mostram valores do writeIndex, readIndex, com as letras W e R

Classe CircularBufferTest

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

Cria CircularBuffer para uso tanto pelo produtor como pelo consumidor

Executa o produtor e o consumidor em threads separadas

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

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

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

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

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

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

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

- Gravou no índice 1, avançou o índice de gravação para posição 2 (W)
- Índice de leitura (R) permanece na posição 1

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

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

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

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

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

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

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

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

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

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

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

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


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


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

Relacionamento produtor/ consumidor: ArrayBlockingQueue

- Versão completamente implementada do buffer circular.
- Implementa a interface BlockingQueue.
- Declara os métodos put e take para gravar dados no buffer e ler dados do buffer, respectivamente.

Classe BlockingBuffer

```
1  // Fig. 23.15: BlockingBuffer.java
2  // Classe sincroniza acesso a um buffer de bloqueio.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer
6  {
7     private ArrayBlockingQueue<Integer> buffer;
8
9     public BlockingBuffer()
10    {
11       buffer = new ArrayBlockingQueue<Integer>( 3 );
12    } // fim do construtor BlockingBuffer
13
14    // coloca o valor no buffer
15    public void set( int value )
16    {
17       try
18       {
19          buffer.put( value ); // coloca o valor no buffer circular
20          System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
21             "Buffers occupied: ", buffer.size() );
22       } // fim do try
23       catch ( Exception exception )
24       {
25          exception.printStackTrace();
26       } // fim do catch
27    } // fim do método set
28
```

Cria uma instância de `ArrayBlockingQueue` para armazenar dados

Coloca um valor no buffer; bloqueia se o buffer estiver cheio

Classe BlockingBuffer

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

Remove o valor do buffer; bloqueia se o buffer estiver vazio

Classe BlockingBufferTest

```
1  // Fig 23.16: BlockingBufferTest.java
2  // Aplicativo mostra duas threads que manipulam um buffer de bloqueio.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class BlockingBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de thread com duas threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // cria BlockingBuffer para armazenar ints
14       Buffer sharedLocation = new BlockingBuffer();
15
16       try // tenta iniciar produra e consumidora
17       {
18          application.execute( new Producer( sharedLocation ) );
19          application.execute( new Consumer( sharedLocation ) );
20       } // fim do try
21       catch ( Exception exception )
22       {
23          exception.printStackTrace();
24       } // fim do catch
25
```

Cria um `BlockingBuffer` para uso no produtor e consumidor

Executa o produtor e o consumidor em threads separadas

Classe BlockingBufferTest

```
26         application.shutdown();
27     } // fim do main
28 } // fim da classe BlockingBufferTest
```

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

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

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

Multithreading com GUI

- Componentes GUI Swing:
  - Não são seguros para threads.
  - As atualizações devem ser realizadas no caso de uma thread de despacho de evento.
    - Utiliza o método static **invokeLater** (trata despacho de thread) da classe **SwingUtilities** e passa para ele um objeto **Runnable**.

Multithreading com GUI

- Exemplo:
  - Cria uma GUI e 3 threads, cada uma apresentando constantemente uma letra aleatória
  - Tem botões para suspender uma thread
    - chama **wait** (suspender)
  - Quando o botão não está "selecionado", chama método para retomar execução da thread - **signal**
  - Usa um array de booleans para guardar quais threads estão suspensas

Classe RunnableObject

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

Implementa a interface Runnable

Lock (bloqueia) para implementar exclusão mútua

Variável Condition para suspender as threads

Boolean para controlar se a thread foi suspensa

Cria um Lock e uma variável Condition

Obtém o nome da thread atual

Classe RunnableObject

```
31        while ( true ) // infinito; será terminado de fora
32        {
33           try
34           {
35              // dorme por até 1 segundo
36              Thread.sleep( generator.nextInt( 1000 ) );
37
38              lockObject.lock(); // obtém o bloqueio
39              try
40              {
41                 while ( suspended ) faz loop até não ser suspenso
42                 {
43                    suspend.await(); // suspende a execução do thread
44                 } // fim do while
45              } // fim do try
46              finally
47              {
48                 lockObject.unlock(); // desbloqueia o bloqueio
49              } // fim do finally
50           } // fim do try
51           // se a thread interrompida durante espera/enquanto dormia
52           catch ( InterruptedException exception )
53           {
54              exception.printStackTrace(); // imprime o rastreamento de pilha
55           } // fim do catch
56
```

Obtém o bloqueio para impor a exclusão mútua

Espera enquanto a thread é suspensa

Libera o bloqueio

Classe RunnableObject

```
57          // exibe o caractere no JLabel correspondente
58          SwingUtilities.invokeLater(
59             new Runnable()
60             {
61                // seleciona o caractere aleatório e o exibe
62                public void run()
63                {
64                   // seleciona a letra maiúscula aleatória
65                   char displayChar =
66                      ( char ) ( generator.nextInt( 26 ) + 65 );
67
68                   // gera saída de caractere em JLabel
69                   output.setText( threadName + ": " + displayChar );
70                } // fim do método run
71             } // fim da classe interna
72          ); // fim da chamada para SwingUtilities.invokeLater
73       } // fim do while
74    } // fim do método run
75
```

Chama `invokeLater`

Uma `Runnable` é passada para o método `invokeLater`

Classe RunnableObject

```
76     // altera o estado suspenso/em execução
77     public void toggle()
78     {
79        suspended = !suspended; // alterna booleano que controla estado
80
81        // muda cor de rótulo na suspensão/retomada
82        output.setBackground( suspended ? Color.RED : Color.GREEN );
83
84        lockObject.lock(); // obtém o bloqueio
85        try
86        {
87           if ( !suspended ) // se a thread foi retomada
88           {
89              suspend.signal(); // libera o bloqueio
90           } // fim do if
91        } // fim do try
92        finally
93        {
94           lockObject.unlock(); // libera o bloqueio
95        } // fim do finally
96     } // fim do método toggle
97  } // fim da classe RunnableObject
```

Obtém o bloqueio para a aplicação

Retoma uma thread em espera

Libera o bloqueio

Classe RandomCharacteres

```
1  // Fig. 23.18: RandomCharacters.java
2  // A classe RandomCharacters demonstra a interface Runnable
3  import java.awt.Color;
4  import java.awt.GridLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import java.util.concurrent.Executors;
8  import java.util.concurrent.ExecutorService;
9  import java.util.concurrent.locks.Condition;
10 import java.util.concurrent.locks.Lock;
11 import java.util.concurrent.locks.ReentrantLock;
12 import javax.swing.JCheckBox;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15
16 public class RandomCharacters extends JFrame implements ActionListener
17 {
18    private final static int SIZE = 3; // número de threads
19    private JCheckBox checkboxes[]; // array de JCheckBoxes
20    private Lock lockObject = new ReentrantLock( true ); // único bloqueio
21
22    // array de RunnableObjects para exibir caracteres aleatórios
23    private RunnableObject[] randomCharacters =
24       new RunnableObject[ SIZE ];
25
```

Cria um Lock para a aplicação

Classe RandomCharacteres

```
26    // configura GUI e arrays
27    public RandomCharacters()
28    {
29       checkboxes = new JCheckBox[ SIZE ]; // aloca espaço para array
30       setLayout( new GridLayout( SIZE, 2, 5, 5 ) ); // configura o layout
31
32       // cria novo pool de threads com threads SIZE
33       ExecutorService runner = Executors.newFixedThreadPool( SIZE );
34
35       // loop itera SIZE vezes
36       for ( int count = 0; count < SIZE; count++ )
37       {
38          JLabel outputJLabel = new JLabel(); // cria JLabel
39          outputJLabel.setBackground( Color.GREEN ); // configura cor
40          outputJLabel.setOpaque( true ); // configura JLabel para ser opaco
41          add( outputJLabel ); // adiciona JLabel ao JFrame
42
43          // cria JCheckBox para controlar suspender/retomar o estado
44          checkboxes[ count ] = new JCheckBox( "Suspended" );
45
46          // adiciona o ouvinte que executa quando JCheckBox é clicada
47          checkboxes[ count ].addActionListener( this );
48          add( checkboxes[ count ] ); // adiciona JCheckBox ao JFrame
49
```

Cria um pool de threads para as threads em execução

Classe RandomCharacteres

```
50             // cria um novo RunnableObject
51             randomCharacters[ count ] =
52                new RunnableObject( lockObject, outputJLabel );
53
54             // executa RunnableObject
55             runner.execute( randomCharacters[ count ] );
56          } // fim do for
57
58          setSize( 275, 90 ); // configura o tamanho da janela
59          setVisible( true ); // configura a janela
60
61          runner.shutdown(); // desliga quando as threads terminam
62       } // fim do construtor RandomCharacters
63
64       // trata eventos da JCheckBox
65       public void actionPerformed( ActionEvent event )
66       {
67          // faz loop sobre todas as JCheckBoxes no array
68          for ( int count = 0; count < checkboxes.length; count++ )
69          {
70             // verifica se essa JCheckBox foi a origem do evento
71             if ( event.getSource() == checkboxes[ count ] )
72                randomCharacters[ count ].toggle(); // alterna o estado
73          } // fim do for
74       } // fim do método actionPerformed
75
```

Executa uma `Runnable`

Desativa o pool de threads quando as threads concluem suas tarefas

Classe RandomCharacteres

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

Exercício

1. Corrida de cavalos simulada

Considere uma aplicação para simular uma corrida de cavalos. Na corrida participam sempre 3 cavalos que correm de forma independente. A evolução de cada cavalo deve ser visível num campo do tipo JTextField, tal como se exemplifica na figura em baixo. A interface da aplicação inclui, ainda, um botão para iniciar a corrida. A movimentação dos cavalos deve obedecer aos seguintes requisitos:

a) o percurso total dos cavalos é constituído por 30 movimentos;

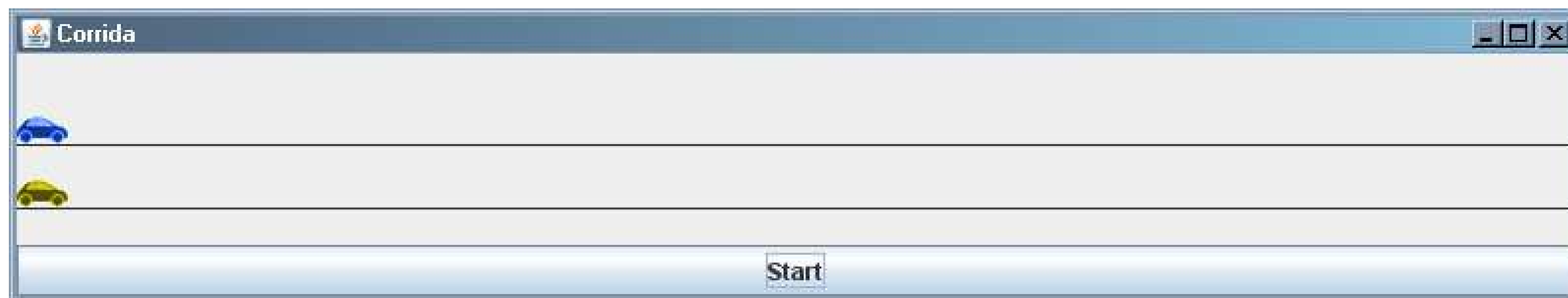
b) um movimento consiste em subtrair uma unidade aos movimentos que faltam;

c) após cada movimento o cavalo deve dormir um tempo aleatório.

Exercício

2. Car Race

Faça um programa que permita simular uma corrida de dois automóveis.



Os automóveis devem ser controlados por dois processos (threads) que vão dando ordens para o automóvel avançar. Os processos devem terminar quando um dos carros chegar ao fim do percurso. A cada avanço o automóvel deve dormir um tempo aleatório