UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Desenvolvimento de Software Concorrente

Multithreading

Slides baseados no Capítulo 26 de Java How to Program, Deitel, 9/E

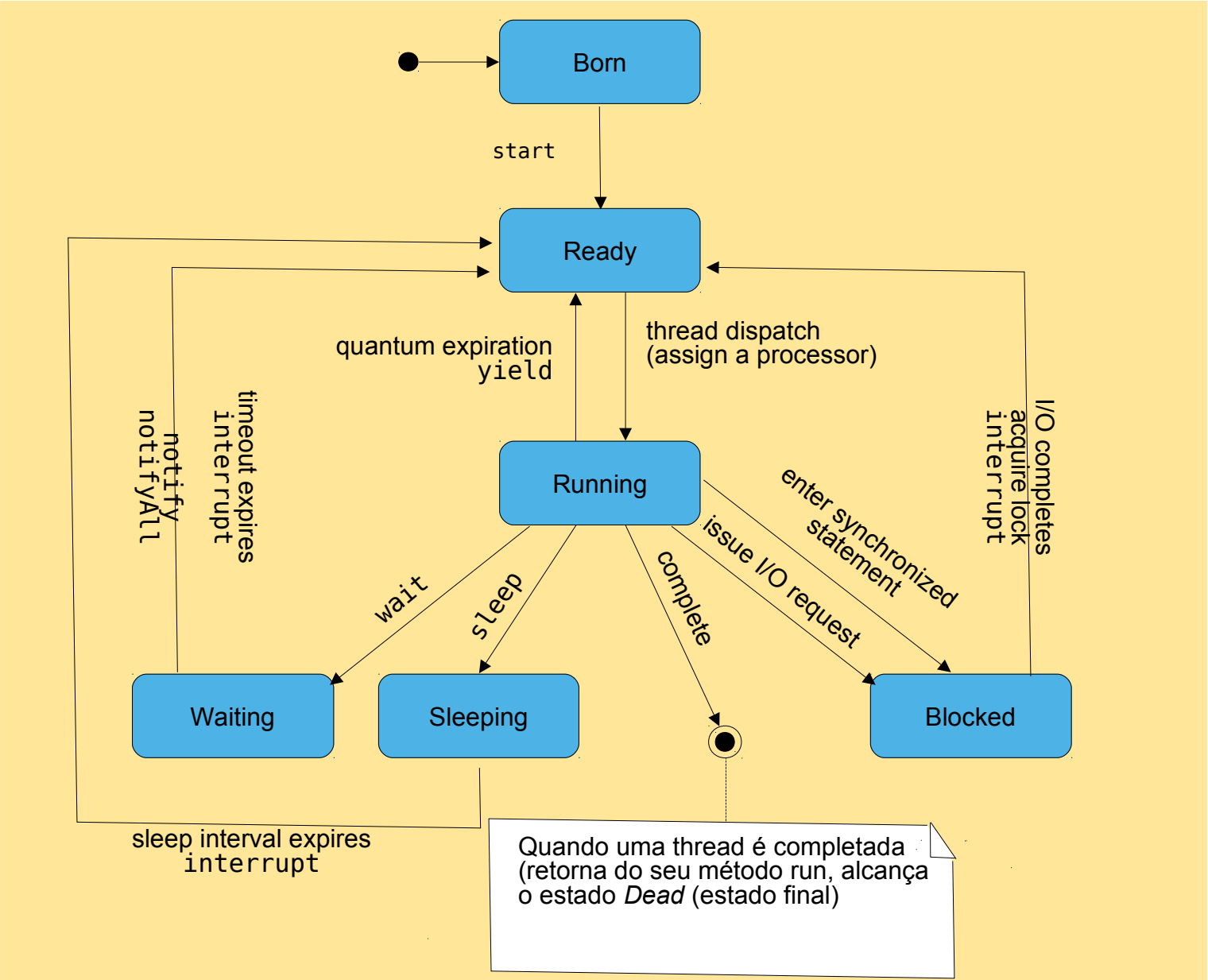
Introdução

- Concorrência normalmente disponível em primitivas do SO
- Java está preparada para multithreading (built-in)
  - Multithreading incrementa o desempenho de alguns programas

Estados de uma Thread: ciclo de vida

- Estados de um thread
  - Inicial (born)
    - Thread foi apenas criada
  - Pronto (ready)
    - Método `start` da Thread invocado
    - Thread pode agora executar
  - Execução (running)
    - Thread está em execução usando um processador
  - Finalizado (dead)
    - Thread completada ou terminada
    - Pode ser desalocada pelo sistema

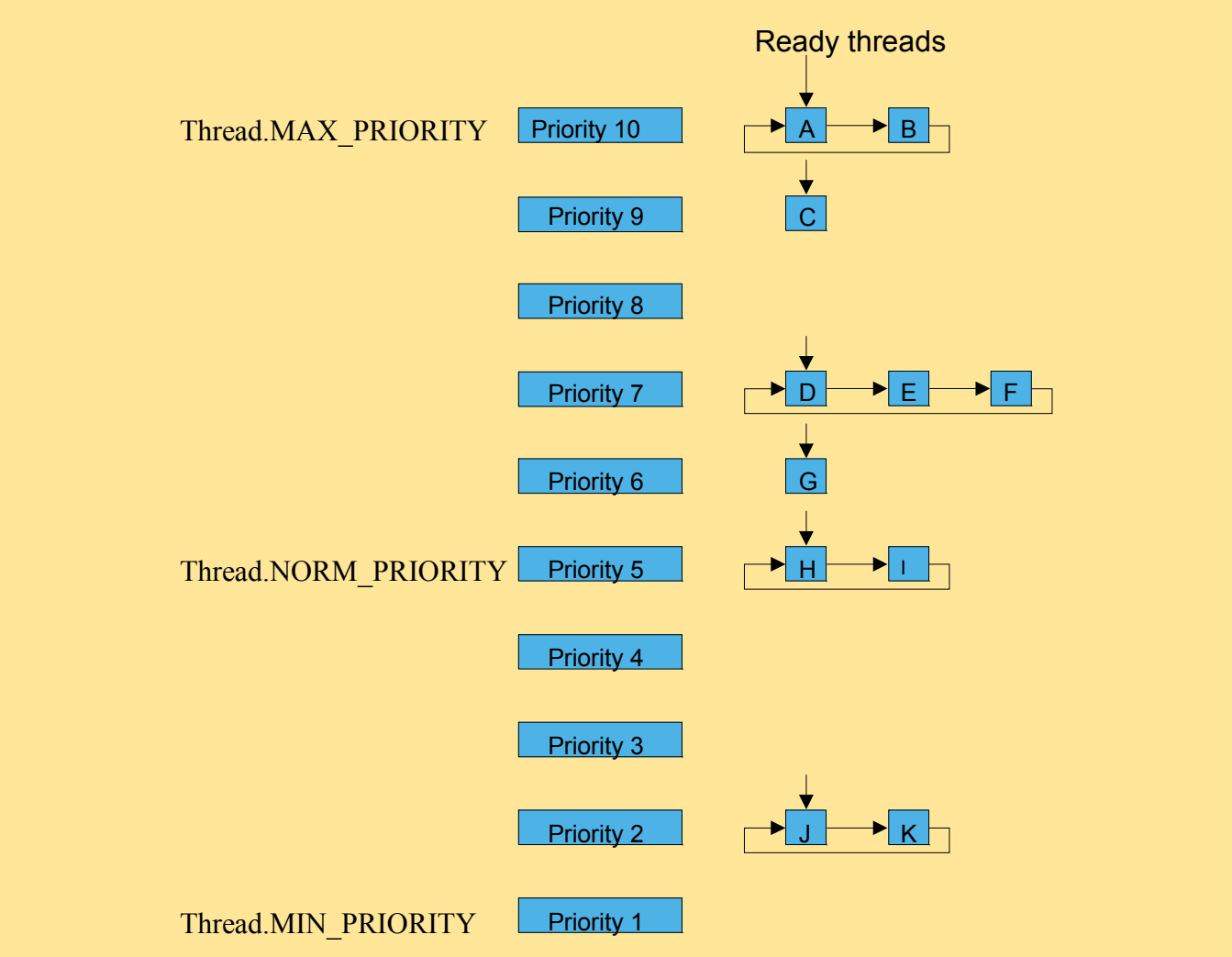
Ciclo de vida de uma Thread



Prioridades e Escalonamento

- Prioridade de uma thread Java
  - Prioridade no arranjo 1-10
- Timeslicing
  - A cada thread é atribuído um tempo de processador (denominado quantum)
  - A thread com a prioridade mais alta permanece em execução

Exemplo de escalonamento de prioridades de uma Thread



Criando e executando Threads

- Estado Sleep

» Método `sleep` da thread é invocado

» Thread dorme por um intervalo de tempo e então acorda

```
1   // Fig. 16.3: ThreadTester.java
2   // Multiple threads printing at different intervals.
3
4   public class ThreadTester {
5
6      public static void main( String [] args )
7      {
8         // create and name each thread
9         PrintThread thread1 = new PrintThread( "thread1" );
10        PrintThread thread2 = new PrintThread( "thread2" );
11        PrintThread thread3 = new PrintThread( "thread3" );
12
13        System.err.println( "Starting threads" );
14
15        thread1.start(); // start thread1 and place it in ready state
16        thread2.start(); // start thread2 and place it in ready state
17        thread3.start(); // start thread3 and place it in ready state
18
19        System.err.println( "Threads started, main ends\n" );
20
21     } // end main
22
23  } // end class ThreadTester
24
```

ThreadTester.java

Lines 9-11

create four PrintThreads

call start methods

```
25 // class PrintThread controls thread execution
26 class PrintThread extends Thread {
27    private int sleepTime;
28
29    // assign name to thread by calling superclass constructor
30    public PrintThread( String name )
31    {
32       super( name );
33
34       // pick random sleep time between 0 and 5 seconds
35       sleepTime = ( int ) ( Math.random() * 5001 );
36    }
37
38    // method run is the code to be executed by new thread
39    public void run()
40    {
41       // put thread to sleep for sleepTime amount of time
42       try {
43          System.err.println(
44             getName() + " going to sleep for " + sleepTime );
45
46          Thread.sleep( sleepTime );
47       }
48
```

ThreadTester.ja

PrintThread extends Thread

Line 26

Line 35

Constructor initializes sleepTime

When the thread enters the running state, run is called

```java
49         // if thread interrupted during sleep, print stack trace
50         catch ( InterruptedException exception ) {
51            exception.printStackTrace();
52         }
53
54         // print thread name
55         System.err.println( getName() + " done sleeping" );
56
57      } // end method run
58
59   } // end class PrintThread
```

```
Starting threads
Threads started, main ends

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

ThreadTester.java

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

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

ThreadTester.java

Sincronização

- Java usa monitores para a sincronização das threads
- Palavra-chave `sychronized`
  - Cada método `synchronized` de um objeto tem um monitor
  - Quando uma thread "está dentro" de um método `synchronized`, todas as demais ficam bloqueadas até a finalização do método
  - Quando o método finaliza, a próxima thread com prioridade mais alta é executada

Produtor/Consumidor sem Sincronização

- Buffer
  - Região de memória compartilhada
- Thread produtora (producer)
  - Gera dados para serem adicionados no buffer
  - Chama `wait` se o consumidor não tiver lido mensagens no buffer
  - Escreve no buffer e chama `notify`
- Thread consumidora
  - Lê dados do buffer
  - Chama `wait` se o buffer estiver vazio
- Sincroniza threads para evitar que os dados sejam corrompidos

```java
// Fig. 16.4: Buffer.java
// Buffer interface specifies methods called by Producer and Consumer.

public interface Buffer {
   public void set( int value );  // place value into Buffer
   public int get();              // return value from Buffer
}
```

Buffer.java

```
1   // Fig. 16.5: Producer.java
2   // Producer's run method controls a thread that
3   // stores values from 1 to 4 in sharedLocation.
4
5   public class Producer extends Thread {
6      private Buffer sharedLocation; // reference to shared object
7
8      // constructor
9      public Producer( Buffer shared )
10     {
11         super( "Producer" );
12         sharedLocation = shared;
13     }
14
15     // store values from 1 to 4 in sharedLocation
16     public void run()
17     {
18        for ( int count = 1; count <= 4; count++ ) {
19
20           // sleep 0 to 3 seconds, then place value in Buffer
21           try {
22              Thread.sleep( ( int ) ( Math.random() * 3001 ) );
23              sharedLocation.set( count );
24           }
25
```

`Producer` extends `Thread`

Line 5

This is a shared object

Line 6

Line 16

Lines 22-23

Method `run` is overridden

The thread goes to sleep, then the buffer is set

```
26          // if sleeping thread interrupted, print stack trace
27          catch ( InterruptedException exception ) {
28             exception.printStackTrace();
29          }
30
31       } // end for
32
33       System.err.println( getName() + " done producing." +
34          "\nTerminating " + getName() + ".");
35
36    } // end method run
37
38 } // end class Producer
```

Producer.java

```
1   // Fig. 16.6: Consumer.java
2   // Consumer's run method controls a thread that loops four
3   // times and reads a value from sharedLocation each time.
4
5   public class Consumer extends Thread {
6      private Buffer sharedLocation; // reference to shared object
7
8      // constructor
9      public Consumer( Buffer shared )
10     {
11        super( "Consumer" );
12        sharedLocation = shared;
13     }
14
15     // read sharedLocation's value four times and sum the values
16     public void run()
17     {
18        int sum = 0;
19
20        for ( int count = 1; count <= 4; count++ ) {
21
22           // sleep 0 to 3 seconds, read value from Buffer and add to sum
23           try {
24              Thread.sleep( ( int ) ( Math.random() * 3001 ) );
25              sum += sharedLocation.get();
26           }
27
```

`Consumer` extends `Thread`

Line 5

This is a shared object

Line 6

Line 16

Lines 24-25

Method `run` is overridden

The thread goes to sleep, then the buffer is read

```
28          // if sleeping thread interrupted, print stack trace
29          catch ( InterruptedException exception ) {
30             exception.printStackTrace();
31          }
32       }
33
34       System.err.println( getName() + " read values totaling: " + sum +
35          ".\nTerminating " + getName() + ".");
36
37    } // end method run
38
39 } // end class Consumer
```

Consumer.java

```
1   // Fig. 16.7: UnsynchronizedBuffer.java
2   // UnsynchronizedBuffer represents a single shared integer.
3
4   public class UnsynchronizedBuffer implements Buffer {
5      private int buffer = -1; // shared by producer and consumer thre
6
7      // place value into buffer
8      public void set( int value )
9      {
10        System.err.println( Thread.currentThread().getName() +
11           " writes " + value );
12
13        buffer = value;
14     }
15
16     // return value from buffer
17     public int get()
18     {
19        System.err.println( Thread.currentThread().getName() +
20           " reads " + buffer );
21
22        return buffer;
23     }
24
25  } // end class UnsynchronizedBuffer
```

This class implements the `Buffer` interface

The data is a single integer

This method sets the value in the buffer

Lines 8 and 13

Lines 17 and 22

This method reads the value in the buffer

```
1   // Fig. 16.8: SharedBufferTest.java
2   // SharedBufferTest creates producer and consumer threads.
3
4   public class SharedBufferTest {
5
6       public static void main( String [] args )
7       {
8           // create shared object used by threads
9           Buffer sharedLocation = new UnsynchronizedBuffer();
10
11          // create producer and consumer objects
12          Producer producer = new Producer( sharedLocation );
13          Consumer consumer = new Consumer( sharedLocation );
14
15          producer.start();  // start producer thread
16          consumer.start();  // start consumer thread
17
18      } // end main
19
20  } // end class SharedCell
```

SharedBufferTest.java

Line 9

Create a Buffer object

Lines 12-13

Create a Producer and a Consumer

Start the Producer and Consumer threads

```
Consumer reads -1
Producer writes 1
Consumer reads 1
Consumer reads 1
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.
```

SharedBufferTest.java

```
Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.
```

```
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.
```

SharedBufferTest.java

Produtor/Consumidor com Sincronização

- Sincroniza threads para garantir dados corretos

```
1   // Fig. 16.9: SynchronizedBuffer.java
2   // SynchronizedBuffer synchronizes access to a single shared inte
3
4   public class SynchronizedBuffer implements Buffer {
5      private int buffer = -1; // shared by producer and consumer th
6      private int occupiedBufferCount = 0; // count of occupied buff
7
8      // place value into buffer
9      public synchronized void set( int value )
10     {
11        // for output purposes, get name of thread that called this method
12        String name = Thread.currentThread().getName();
13
14        // while there are no empty locations, place thread in waiting state
15        while ( occupiedBufferCount == 1 ) {
16
17           // output thread information and buffer information, the
18           try {
19              System.err.println( name + " tries to write." );
20              displayState( "Buffer full. " + name + " waits." );
21              wait();
22           }
23
24           // if waiting thread interrupted, print stack trace
25           catch ( InterruptedException exception ) {
26              exception.printStackTrace();
27           }
```

This class implements the `Buffer` interface

SynchronizedBuf

Remember the number of filled spaces

Method `set` is declared `synchronized`

Line 9

Get the name of the thread

Line 12

Wait while the buffer is filled

```
28
29        } // end while
30
31        buffer = value; // set new buffer value
32
33        // indicate producer cannot store another value
34        // until consumer retrieves current buffer value
35        ++occupiedBufferCount;
36
37        displayState( name + " writes " + buffer );
38
39        notify(); // tell waiting thread to enter ready state
40
41     } // end method set; releases lock on SynchronizedBuffer
42
43     // return value from buffer
44     public synchronized int get()
45     {
46        // for output purposes, get name of thread that called this method
47        String name = Thread.currentThread().getName();
48
```

Write to the buffer

hronizedBuf
fer.java

Increment the buffer count

Line 35

Alert a waiting thread

Line 39

Line 44

Method `get` is declared `synchronized`

Get the name of the thread

```
49        // while no data to read, place thread in waiting state
50        while ( occupiedBufferCount == 0 ) {
51
52           // output thread information and buffer information, then wait
53           try {
54              System.err.println( name + " tries to read." );
55              displayState( "Buffer empty. " + name + " waits." );
56              wait();
57           }
58
59           // if waiting thread interrupted, print stack trace
60           catch ( InterruptedException exception ) {
61              exception.printStackTrace();
62           }
63
64        } // end while
65
66        // indicate that producer can store another value
67        // because consumer just retrieved buffer value
68        --occupiedBufferCount;
69
70        displayState( name + " reads " + buffer );
71
72        notify(); // tell waiting thread to become ready to execute
73
74        return buffer;
```

Wait while the buffer is empty

Lines 50 and 56

Line 68

Line 72

Line 74

Decrement the buffer count

Alert a waiting thread

Return the buffer

```java

   } // end method get; releases lock on SynchronizedBuffer

   // display current operation and buffer state
   public void displayState( String operation )
   {
      StringBuffer outputLine = new StringBuffer( operation );
      outputLine.setLength( 40 );
      outputLine.append( buffer + "\t\t" + occupiedBufferCount );
      System.err.println( outputLine );
      System.err.println();
   }

} // end class SynchronizedBuffer
```

SynchronizedBuffer.java

```
1   // Fig. 16.10: SharedBufferTest2.java
2   // SharedBufferTest2creates producer and consumer threads.
3
4   public class SharedBufferTest2 {
5
6      public static void main( String [] args )
7      {
8         // create shared object used by threads; we use a SynchronizedBuffer
9         // reference rather than a Buffer reference so we can invoke
10        // SynchronizedBuffer method displayState from main
11        SynchronizedBuffer sharedLocation = new SynchronizedBuffer();
12
13        // Display column heads for output
14        StringBuffer columnHeads = new StringBuffer( "Operation" );
15        columnHeads.setLength( 40 );
16        columnHeads.append( "Buffer\t\tOccupied Count" );
17        System.err.println( columnHeads );
18        System.err.println();
19        sharedLocation.displayState( "Initial State" );
20
21        // create producer and consumer objects
22        Producer producer = new Producer( sharedLocation );
23        Consumer consumer = new Consumer( sharedLocation );
24
```

SharedBufferTest2.java

Create a Buffer object

Line 11

Line 19

Lines 22-23

Output initial state

Create a Producer and a Consumer

```
25        producer.start();  // start producer thread
26        consumer.start();  // start consumer thread
27
28     } // end main
29
30  } // end class SharedBufferTest2
```

Start the `Producer` and `Consumer` threads

`SharedBufferTest2.java`

Lines 25-26

```
Operation                                Buffer          Occupied Count

Initial State                            -1              0

Consumer tries to read.
Buffer empty. Consumer waits.            -1              0

Producer writes 1                        1               1

Consumer reads 1                         1               0

Consumer tries to read.
Buffer empty. Consumer waits.            1               0

Producer writes 2                        2               1

Consumer reads 2                         2               0

Producer writes 3                        3               1
```

```
Consumer reads 3                        3               0

Consumer tries to read.
Buffer empty. Consumer waits.           3               0

Producer writes 4                       4               1

Consumer reads 4                        4               0
Producer done producing.
Terminating Producer.

Consumer read values totaling: 10.
Terminating Consumer.
```

SharedBufferTest2.java

```
Operation                               Buffer          Occupied Count

Initial State                           -1              0

Consumer tries to read.
Buffer empty. Consumer waits.           -1              0

Producer writes 1                       1               1

Consumer reads 1                        1               0

Producer writes 2                       2               1
```

```
Producer tries to write.
Buffer full. Producer waits.            2               1

Consumer reads 2                        2               0

Producer writes 3                       3               1

Consumer reads 3                        3               0

Producer writes 4                       4               1

Producer done producing.
Terminating Producer.
Consumer reads 4                        4               0

Consumer read values totaling: 10.
Terminating Consumer.
```

SharedBufferTest2.java

```
Operation                               Buffer          Occupied Count

Initial State                           -1              0

Producer writes 1                       1               1

Consumer reads 1                        1               0

Producer writes 2                       2               1
```

```
Consumer reads 2                        2               0

Producer writes 3                       3               1

Consumer reads 3                        3               0

Producer writes 4                       4               1

Producer done producing.
Terminating Producer.
Consumer reads 4                        4               0

Consumer read values totaling: 10.
Terminating Consumer.
```

SharedBufferTest2.java

Produtor/Consumidor: Buffer Circular

- Buffer Circular
  - Múltiplas células de memória
  - Produz um item se uma ou mais células estiverem vazias
  - Consome um item se uma ou mais células estiverem cheias

```
1  // Fig. 16.11: RunnableOutput.java
2  // Class RunnableOutput updates JTextArea with output
3  import javax.swing.*;
4
5  public class RunnableOutput implements Runnable {
6     private JTextArea outputArea;
7     private String messageToAppend;
8
9     // initialize outputArea and message
10    public RunnableOutput( JTextArea output, String message )
11    {
12       outputArea = output;
13       messageToAppend = message;
14    }
15
16    // method called by SwingUtilities.invokeLater to update outputArea
17    public void run()
18    {
19       outputArea.append( messageToAppend );
20    }
21
22 } // end class RunnableOutput
```

This class implements the `Runnable` interface

leOutput.

Line 5

Line 17

Declare method `run`

```
1   // Fig. 16.12: Producer.java
2   // Producer's run method controls a thread that
3   // stores values from 11 to 20 in sharedLocation.
4   import javax.swing.*;
5
6   public class Producer extends Thread {
7      private Buffer sharedLocation;
8      private JTextArea outputArea;
9
10     // constructor
11     public Producer( Buffer shared, JTextArea output )
12     {
13        super( "Producer" );
14        sharedLocation = shared;
15        outputArea = output;
16     }
17
18     // store values from 11-20 and in sharedLocation's buffer
19     public void run()
20     {
21        for ( int count = 11; count <= 20; count ++ ) {
22
23           // sleep 0 to 3 seconds, then place value in Buffer
24           try {
25              Thread.sleep( ( int ) ( Math.random() * 3000 ) );
26              sharedLocation.set( count );
27           }
```

Producer.java

Lines 21 and 26

Write the values from 11 to 20 to the buffer

```
28
29          // if sleeping thread interrupted, print stack trace
30          catch ( InterruptedException exception ) {
31             exception.printStackTrace();
32          }
33       }
34
35       String name = getName();
36       SwingUtilities.invokeLater( new RunnableOutput( outputArea, "\n" +
37          name + " done producing.\n" + name + " terminated.\n" ) );
38
39    } // end method run
40
41 } // end class Producer
```

Producer.java

Lines 36-37

Update the output

Consumer.java

```java
// Fig. 16.13: Consumer.java
// Consumer’s run method controls a thread that loops ten
// times and reads a value from sharedLocation each time.
import javax.swing.*;

public class Consumer extends Thread {
   private Buffer sharedLocation; // reference to shared object
   private JTextArea outputArea;

   // constructor
   public Consumer( Buffer shared, JTextArea output )
   {
      super( "Consumer" );
      sharedLocation = shared;
      outputArea = output;
   }

   // read sharedLocation's value ten times and sum the values
   public void run()
   {
      int sum = 0;

      for ( int count = 1; count <= 10; count++ ) {

```

```
25          // sleep 0 to 3 seconds, read value from Buffer and add to sum
26          try {
27             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
28             sum += sharedLocation.get();
29          }
30
31          // if sleeping thread interrupted, print stack trace
32          catch ( InterruptedException exception ) {
33             exception.printStackTrace();
34          }
35       }
36
37       String name = getName();
38       SwingUtilities.invokeLater( new RunnableOutput( outputArea,
39          "\nTotal " + name + " consumed: " + sum + ".\n" +
40          name + " terminated.\n ") );
41
42    } // end method run
43
44 } // end class Consumer
```

Read the value from the buffer

er.java

Line 28

Lines 38-40

Update the output

```java
// Fig. 16.14: CircularBuffer.java
// CircularBuffer synchronizes access to an array of shared buffers.
import javax.swing.*;

public class CircularBuffer implements Buffer {

   // each array element is a buffer
   private int buffers[] = { -1, -1, -1 };

   // occupiedBufferCount maintains count of occupied buffers
   private int occupiedBufferCount = 0;

   // variables that maintain read and write buffer locations
   private int readLocation = 0, writeLocation = 0;

   // reference to GUI component that displays output
   private JTextArea outputArea;

   // constructor
   public CircularBuffer( JTextArea output )
   {
      outputArea = output;
   }

```

CircularBuffer.
java

The data is an array of three integers

Remember the number of filled spaces

Remember the read and write positions

```
25     // place value into buffer
26     public synchronized void set( int value )
27     {
28        // for output purposes, get name of thread that called this method
29        String name = Thread.currentThread().getName();
30
31        // while there are no empty locations, place thread in waiting state
32        while ( occupiedBufferCount == buffers.length ) {
33
34           // output thread information and buffer information, then wait
35           try {
36              SwingUtilities.invokeLater( new RunnableOutput( outputArea,
37                 "\nAll buffers full. " + name + " waits." ) );
38              wait();
39           }
40
41           // if waiting thread interrupted, print stack trace
42           catch ( InterruptedException exception )
43           {
44              exception.printStackTrace();
45           }
46
47        } // end while
48
49        // place value in writeLocation of buffers
50        buffers[ writeLocation ] = value;
51
```

Method `set` is declared `synchronized`

`CircularBuffer.java`

Check if the buffer is full

Line 32

Lines 36-37

Line 50

Update the output

Write to the buffer

```
52        // update Swing GUI component with produced value
53        SwingUtilities.invokeLater( new RunnableOutput( outputArea,
54           "\n" + name + " writes " + buffers[ writeLocation ] + " ") );
55
56        // just produced a value, so increment number of occupied buffers
57        ++occupiedBufferCount;
58
59        // update writeLocation for future write operation
60        writeLocation = ( writeLocation + 1 ) % buffers.length;
61
62        // display contents of shared buffers
63        SwingUtilities.invokeLater( new RunnableOutput(
64           outputArea, createStateOutput() ) );
65
66        notify(); // return waiting thread (if there is one) to ready state
67
68     } // end method set
69
70     // return value from buffer
71     public synchronized int get()
72     {
73        // for output purposes, get name of thread that called this method
74        String name = Thread.currentThread().getName();
75
```

Update the output

Increment the buffer count

Update the write location

Update the output

Alert a waiting thread

Method `get` is declared `synchronized`

`CircularBuffer.`

Lines 53-54

Line 57

Line 66

```
76        // while no data to read, place thread in waiting state
77        while ( occupiedBufferCount == 0 ) {

78
79           // output thread information and buffer information, then wait
80           try {
81              SwingUtilities.invokeLater( new RunnableOutput( outputArea,
82                 "\nAll buffers empty. " + name + " waits.") );
83              wait();
84           }
85
86           // if waiting thread interrupted, print stack trace
87           catch ( InterruptedException exception ) {
88              exception.printStackTrace();
89           }
90
91        } // end while
92
93        // obtain value at current readLocation
94        int readValue = buffers[ readLocation ];
95
96        // update Swing GUI component with consumed value
97        SwingUtilities.invokeLater( new RunnableOutput( outputArea,
98           "\n" + name + " reads " + readValue + " ") );
99
100       // just consumed a value, so decrement number of occupied buffers
101       --occupiedBufferCount;
102
```

Check if the buffer is empty

Update the output

Read a value from the buffer

Update the output

Decrement the buffer count

CircularBuffer.
java

Line 77

Line 94

Lines 97-98

Line 101

```
103        // update readLocation for future read operation
104        readLocation = ( readLocation + 1 ) % buffers.length;

105
106        // display contents of shared buffers
107        SwingUtilities.invokeLater( new RunnableOutput(
108           outputArea, createStateOutput() ) );
109
110        notify(); // return waiting thread (if there is one) to ready state
111
112        return readValue;
113
114     } // end method get
115
116     // create state output
117     public String createStateOutput()
118     {
119        // first line of state information
120        String output =
121           "(buffers occupied: " + occupiedBufferCount + ")\nbuffers: ";
122
123        for ( int i = 0; i < buffers.length; i++ )
124           output += " " + buffers[ i ] + "  ";
125
126        // second line of state information
127        output += "\n         ";
128
```

Update the read location

Update the output

Alert a waiting thread

Return a value from the buffer

Line 110

Line 112

```java
      for ( int i = 0; i < buffers.length; i++ )
         output += "---- ";

      // third line of state information
      output += "\n         ";

      // append readLocation (R) and writeLocation (W)
      // indicators below appropriate buffer locations
      for ( int i = 0; i < buffers.length; i++ )

         if ( i == writeLocation && writeLocation == readLocation )
            output += " WR  ";
         else if ( i == writeLocation )
            output += " W   ";
         else if ( i == readLocation )
            output += "  R  ";
         else
            output += "     ";

      output += "\n";

      return output;

   } // end method createStateOutput

} // end class CircularBuffer
```

CircularBuffer.
java

```
1   // Fig. 16.15: CircularBufferTest.java
2   // CircularBufferTest shows two threads manipulating a circular buffer.
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6
7   // set up the producer and consumer threads and start them
8   public class CircularBufferTest extends JFrame {
9      JTextArea outputArea;
10
11     // set up GUI
12     public CircularBufferTest()
13     {
14        super( "Demonstrating Thread Synchronizaton" );
15
16        outputArea = new JTextArea( 20,30 );
17        outputArea.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
18        getContentPane().add( new JScrollPane( outputArea ) );
19
20        setSize( 310, 500 );
21        setVisible( true );
22
23        // create shared object used by threads; we use a CircularBuffer
24        // reference rather than a Buffer reference so we can invoke
25        // CircularBuffer method createStateOutput
26        CircularBuffer sharedLocation = new CircularBuffer( outputArea );
27
```

CircularBufferTest.java

Line 26

Create a Buffer object

```
28        // display initial state of buffers in CircularBuffer
29        SwingUtilities.invokeLater( new RunnableOutput( outputArea,
30           sharedLocation.createStateOutput() ) );
31
32        // set up threads
33        Producer producer = new Producer( sharedLocation, outputArea );
34        Consumer consumer = new Consumer( sharedLocation, outputArea );
35
36        producer.start();  // start producer thread
37        consumer.start();  // start consumer thread
38
39     } // end constructor
40
41     public static void main ( String args[] )
42     {
43        CircularBufferTest application = new CircularBufferTest();
44        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
45     }
46
47  } // end class CirclularBufferTest
```

Update the output

CircularBufferT

Create a `Producer` and a `Consumer`

Start the `Producer` and `Consumer` threads

Lines 36-37

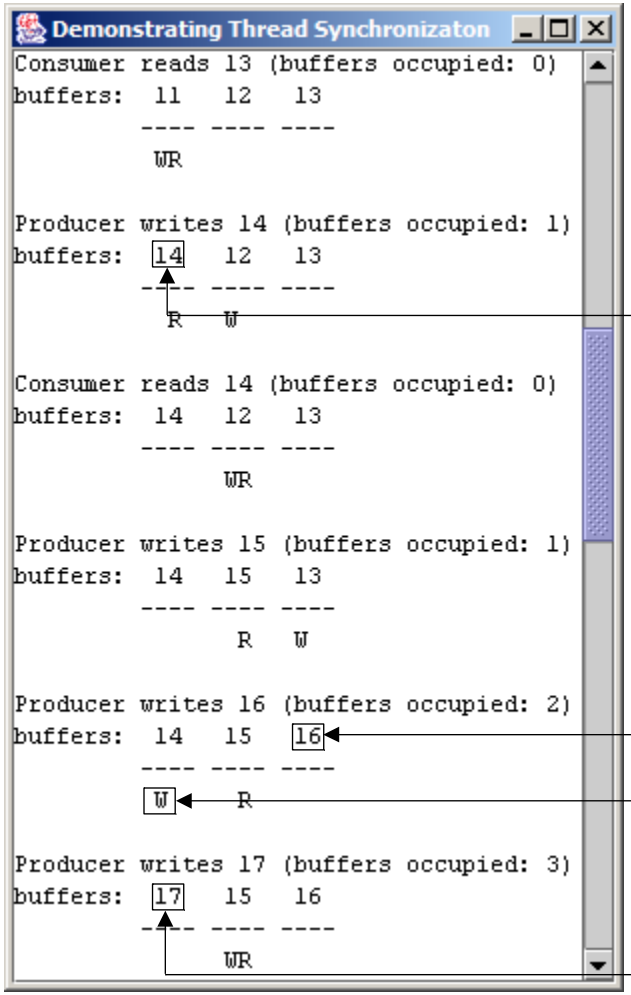
Create an instance of this class

```
Demonstrating Thread Synchronizaton
(buffers occupied: 0)
buffers:  -1   -1   -1
         ---- ---- ----
          WR

All buffers empty. Consumer waits.
Producer writes 11 (buffers occupied: 1)
buffers:  11   -1   -1
         ---- ---- ----
           R   W

Consumer reads 11 (buffers occupied: 0)
buffers:  11   -1   -1
         ---- ---- ----
               WR

All buffers empty. Consumer waits.
Producer writes 12 (buffers occupied: 1)
buffers:  11   12   -1
         ---- ---- ----
                R   W

Consumer reads 12 (buffers occupied: 0)
buffers:  11   12   -1
         ---- ---- ----
                     WR

All buffers empty. Consumer waits.
Producer writes 13 (buffers occupied: 1)
buffers:  11   12   13
         ---- ---- ----
          W          R
```

Value placed in last buffer. Next value will be deposited in first buffer.

CircularBufferTest.java

CircularBufferTest.java



Circular buffer effect—the fourth value is deposited in the first buffer.

Value placed in last buffer. Next value will be deposited in first buffer.

Circular buffer effect—the seventh value is deposited in the first buffer.

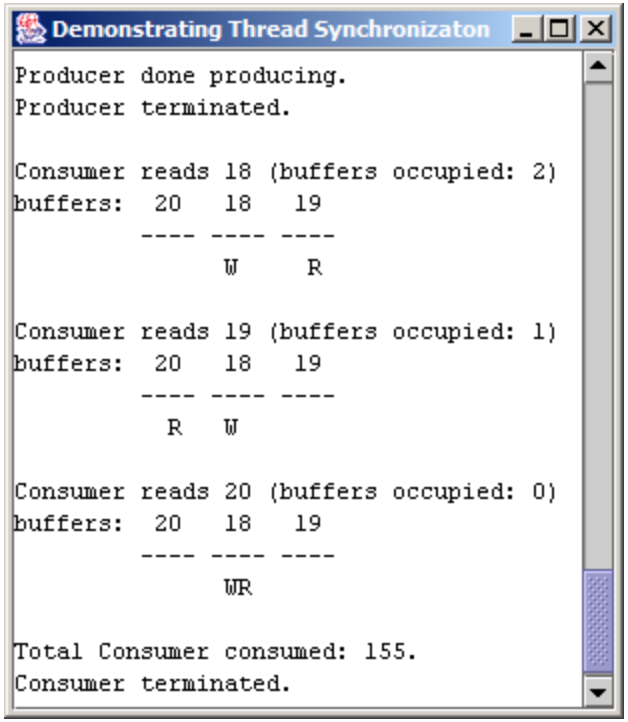
```
Demonstrating Thread Synchronizaton
All buffers full. Producer waits.
Consumer reads 15 (buffers occupied: 2)
buffers:  17   15   16
         ---- ---- ----
               W     R

Producer writes 18 (buffers occupied: 3)
buffers:  17   18   16
         ---- ---- ----
                    WR

All buffers full. Producer waits.
Consumer reads 16 (buffers occupied: 2)
buffers:  17   18   16
         ---- ---- ----
           R        W

Producer writes 19 (buffers occupied: 3)
buffers:  17   18   19
         ---- ---- ----
          WR

All buffers full. Producer waits.
Consumer reads 17 (buffers occupied: 2)
buffers:  17   18   19
         ---- ---- ----
          W     R

Producer writes 20 (buffers occupied: 3)
buffers:  20   18   19
         ---- ---- ----
               WR
```

Value placed in last buffer. Next value will be deposited in first buffer.

Circular buffer effect—the tenth value is deposited in the first buffer.

CircularBufferTest.java

CircularBufferTest.java



16.9 Daemon Threads

- Run for benefit of other threads
  - Do not prevent program from terminating
  - Garbage collector is a daemon thread
- Set daemon thread with method `setDaemon`

`Interface Runnable`

- Uma classe não pode estender mais de uma classe
- `Runnable` pode ser implementada para apoiar multithreading