Chapter 6 – Concurrent Programming

Outline

6.1 Introduction
6.2 Monitors
6.2.1 Condition Variables
6.2.2 Simple Resource Allocation with Monitors
6.2.3 Monitor Example: Circular Buffer
6.2.4 Monitor Example: Readers and Writers
6.3 Java Monitors
6.4 Java Multithreading Case Study, Part III:
   Producer/Consumer Relationship in Java
6.5 Java Multithreading Case Study, Part IV:
   Circular Buffer in Java

Objectives

• After reading this chapter, you should understand:
  – how monitors synchronize access to data.
  – how condition variables are used with monitors.
  – solutions for classic problems in concurrent programming such as readers and writers and circular buffer.
  – Java monitors.
  – remote procedure calls.
6.2.1 Introduction

- Recent interest in concurrent programming languages
  - Naturally express solutions to inherently parallel problems
  - Due to proliferation of multiprocessing systems, distributed systems and massively parallel architectures
  - More complex than standard programs
    - More time required to write, test and debug

6.2 Monitors

- Monitor
  - Contains data and procedures needed to allocate shared resources
    - Accessible only within the monitor
    - No way for threads outside monitor to access monitor data
6.2 Monitors

- Resource allocation using monitors
  - Thread must call monitor entry routine
  - Mutual exclusion is rigidly enforced at monitor boundary
  - A thread that tries to enter monitor when it is in use must wait

- Threads return resources through monitors as well
  - Monitor entry routine calls signal
    - Alerts one waiting thread to acquire resource and enter monitor
  - Higher priority given to waiting threads than ones newly arrived
    - Avoids indefinite postponement
6.2.1 Condition Variables

- Before a thread can reenter the monitor, the thread calling `signal` must first exit monitor
  - Signal-and-exit monitor
    - Requires thread to exit the monitor immediately upon signaling
- Signal-and-continue monitor
  - Allows thread inside monitor to signal that the monitor will soon become available
  - Still maintain lock on the monitor until thread exits monitor
  - Thread can exit monitor by waiting on a condition variable or by completing execution of code protected by monitor

6.2.2 Simple Resource Allocation with Monitors

- Thread inside monitor may need to wait outside until another thread performs an action inside monitor
- Monitor associates separate condition variable with distinct situation that might cause thread to wait
  - Every condition variable has an associated queue
6.2.2 Simple Resource Allocation with Monitors

Figure 6.1 Simple resource allocation with a monitor in pseudocode.

```java
// Fig. 6.1: Resource allocator monitor
// monitor initialization (performed only once)
boolean inUse = false; // simple state variable
Condition available; // condition variable

// request resource
monitorEntry void getResource()
{
  if (!inUse) // is resource in use?
  {
    wait(available); // wait until available is signaled
  } // end if

  inUse = true; // indicate resource is now in use

  // end getResource

// return resource
monitorEntry void returnResource()
{
  inUse = false; // indicate resource is not in use

  signal(available); // signal a waiting thread to proceed
  // end returnResource
```

6.2.3 Monitor Example: Circular Buffer

- Circular buffer implementation of the solution to producer/consumer problem
  - Producer deposits data in successive elements of array
  - Consumer removes the elements in the order in which they were deposited (FIFO)
  - Producer can be several items ahead of consumer
  - If the producer fills last element of array, it must “wrap around” and begin depositing data in the first element of array
6.2.3 Monitor Example: Circular Buffer

- Due to the fixed size of a circular buffer
  - Producer will occasionally find all array elements full, in which case the producer must wait until consumer empties an array element
  - Consumer will occasionally find all array elements empty, in which case the consumer must wait until producer deposits data into an array element

Figure 6.2 Monitor pseudocode implementation of a circular buffer. (Part 1 of 2.)
6.2.3 Monitor Example: Circular Buffer

**Figure 6.2** Monitor pseudocode implementation of a circular buffer. (Part 2 of 2.)

```java
26 // monitor entry called by consumer to read data
27 monitorEntry void getChar( outputParameter slotData )
28 {
29    // wait on condition variable hasData if the buffer is empty
30    if ( occupiedSlots == 0 )
31    {
32        wait( hasData ); // wait until hasData is signaled
33    } // end if
34
35    // read character from buffer into output parameter slotData
36    slotData = circularBuffer[ readPosition ];
37    occupiedSlots--; // one fewer slots has data
38    readerPosition = (readerPosition + 1) % BUFFER_SIZE;
39    signal( hasSpace ); // signal that character has been read
40 } // end getChar
```

6.2.4 Monitor Example: Readers and Writers

- **Readers**
  - Read data, but do not change contents of data
  - Multiple readers can access the shared data at once
  - When a new reader calls a monitor entry routine, it is allowed to proceed as long as no thread is writing and no writer thread is waiting to write
  - After the reader finishes reading, it calls a monitor entry routine to signal the next waiting reader to proceed, causing a “chain reaction”

- **Writers**
  - Can modify data
  - Must have exclusive access
6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 1 of 3.)

```java
// Fig. 6.3: Readers/writers problem
int readers = 0; // number of readers
boolean writeLock = false; // true if a writer is writing
Condition canWrite; // condition variable
Condition canRead; // condition variable

// monitor entry called before performing read
monitorEntry void beginRead()
{
    // wait outside monitor if writer is currently writing or if
    // writers are currently waiting to write
    if (writeLock || queue(canWrite))
    {
        wait(canRead); // wait until reading is allowed
    } // end if
    ++readers; // there is another reader
    signal(canRead); // allow waiting readers to proceed
} // end beginRead

// monitor entry called after reading
monitorEntry void endRead()
{
    --readers; // there are one fewer readers
    // if no more readers are reading, allow a writer to write
    if (readers == 0)
    {
        signal(canWrite); // allow a writer to proceed
    } // end if
} // end endRead

// monitor entry called before performing write
monitorEntry void beginWrite()
{
    // wait if readers are reading or if a writer is writing
    if (readers > 0 || writeLock)
    {
        wait(canWrite); // wait until writing is allowed
    } // end if
}
```

© 2004 Deitel & Associates, Inc. All rights reserved.

6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 2 of 3.)
6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 3 of 3.)

```java
45 writeLock = true; // lock out all readers and writers
46 } // end beginWrite
47 // monitor entry called after performing write
48 monitorEntry void endWrite()
49 {
50 writeLock = false; // release lock
51 // if a reader is waiting to enter, signal a reader
52 if ( queue( canRead ) )
53 {
54 signal( canRead ); // cascade in waiting readers
55 if ( queue( canWrite ) )
56 signal( canWrite ); // one waiting writer can proceed
57 } // end if
58 else // signal a writer if no readers are waiting
59 {
60 signal( canWrite ); // one waiting writer can proceed
61 } // end else
62 } // end endWrite
```

6.3 Java Monitors

- **Java monitors**
  - Primary mechanism for providing mutual exclusion and synchronization in multithreaded Java applications
  - Signal-and-continue monitors
    - Allow a thread to signal that the monitor will soon become available
    - Maintain a lock on monitor until thread exits monitor

- **Keyword synchronized**
  - Imposes mutual exclusion on an object in Java
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

• Java `wait` method
  – Calling thread releases lock on monitor
    • Waits for condition variable
  – After calling `wait`, thread is placed in wait set
    • Thread remains in wait set until signaled by another thread

• Condition variable is implicit in Java
  – A thread may be signaled, reenter the monitor and find that the condition on which it waited has not been met

Figure 6.4 SynchronizedBuffer synchronizes access to a shared integer. (Part 1 of 4.)
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 SynchronizedBuffer synchronizes access to a shared integer. (Part 2 of 4.)

```java
// if waiting thread interrupted, print stack trace
catch (InterruptedException exception)
{
    exception.printStackTrace();
} // end catch

while (buffer = value // set new buffer value
    // indicate producer cannot store another value
    // until consumer retrieves current buffer value
    ++occupiedBuffers;
    displayState(name + " writes " + buffer);
    notify(); // tell waiting thread to enter ready state
} // end method set: releases lock on SynchronizedBuffer
```

© 2004 Deitel & Associates, Inc. All rights reserved.

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 SynchronizedBuffer synchronizes access to a shared integer. (Part 3 of 4.)

```java
// return value from buffer
public synchronized int get()
{
    // for display, get name of thread that called this method
    String name = Thread.currentThread().getName();
    // while no data to read, place thread in waiting state
    while (occupiedBuffers == 0)
    {
        // output thread and buffer information, then wait
        try {
            System.err.println(name + " tries to read.");
            displayState("buffer empty. " + name + " waits.");
        }
        catch (InterruptedException exception)
        {
            exception.printStackTrace();
        } // end catch
    } // end while
```
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

**Figure 6.4** synchronizedBuffer synchronizes access to a shared integer. (Part 4 of 4.)

```java
70 // indicate that producer can store another value
71 // because consumer just retrieved buffer value
72 --occupiedBuffers;
73 displayState( name + " reads " + buffer );
74 notify(); // tell waiting thread to become ready
75 return buffer;
76 } // end method get; releases lock on SynchronizedBuffer
77 // display current operation and buffer state
78 public void displayState( String operation )
79 {
80    StringBuffer outputLine = new StringBuffer( operation );
81    outputLine.setLength( 40 );
82    outputLine.append( buffer + "\t" + occupiedBuffers );
83    System.err.println( outputLine );
84    System.err.println();
85 } // end method displayState
86
87 } // end class SynchronizedBuffer
```

© 2004 Deitel & Associates, Inc. All rights reserved.

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

**Figure 6.5** Threads modifying a shared object with synchronization. (Part 1 of 8.)

```java
1 // Fig. 6.5: SharedBufferTest2.java
2 // SharedBufferTest2 creates producer and consumer threads.
3 public class SharedBufferTest2
4 {
5     public static void main( String [] args )
6     {
7         // create shared object used by threads
8         SynchronizedBuffer sharedLocation = new SynchronizedBuffer();
9         // Display column heads for output
10         StringBuffer columnHeads =
11             new StringBuffer( "Operation" );
12         columnHeads.setLength( 40 );
13         columnHeads.append( "Buffer\tOccupied Count" );
14         System.err.println( columnHeads );
15         System.err.println();
16         sharedLocation.displayState( "Initial State" );
17         // create producer and consumer objects
18         Producer producer = new Producer( sharedLocation );
19         Consumer consumer = new Consumer( sharedLocation );
20         producer.start(); // start producer thread
21         consumer.start(); // start consumer thread
```

© 2004 Deitel & Associates, Inc. All rights reserved.
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 2 of 8.)

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

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 3 of 8.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 4 of 8.)

Figure 6.5 Threads modifying a shared object with synchronization. (Part 5 of 8.)

© 2004 Deitel & Associates, Inc. All rights reserved.
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 6 of 8.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Producer done producing,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminating Producer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Consumer read values totaling</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Terminating Consumer.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sample Output B.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

© 2004 Deitel & Associates, Inc. All rights reserved.

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 7 of 8.)

Sample Output B (Cont.).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Producer done producing, Terminating Producer.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2004 Deitel & Associates, Inc. All rights reserved.
6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 8 of 8.)

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

- Threads issue signals via methods notify or notifyAll
  - notify method
    - wakes one thread waiting to enter the monitor
  - notifyAll method
    - Wakes all waiting threads
6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 SynchronizedBuffer controls access to a shared array of integers. (Part 1 of 7.)

```java
public class CircularBuffer implements Buffer {
    // each array element is a buffer
    private int buffers[] = { -1, -1, -1 };

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

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

    // place value into buffer
    public synchronized void set( int value )
    {
        // get name of thread that called this method
        String name = Thread.currentThread().getName();
```

© 2004 Deitel & Associates, Inc. All rights reserved.

---

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 SynchronizedBuffer controls access to a shared array of integers. (Part 2 of 7.)

```java
    // while buffer full, place thread in waiting state
    while ( occupiedBuffers == buffers.length )
    {
        // output thread and buffer information, then wait
        try {
            System.err.println( "an all buffers full." +
                                name + " waits."");
            wait(); // wait until space is available
        } catch (InterruptedException exception )
        {
            exception.printStackTrace();
        } // end catch
    } // end while

    // place value in writeLocation of buffers
    buffers[ writeLocation ] = value;

    // output produced value
    System.err.println( "\n" + name + " writes " +
                        buffers[ writeLocation ] + "\n" );
```

© 2004 Deitel & Associates, Inc. All rights reserved.
6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 SynchronizedBuffer controls access to a shared array of integers. (Part 3 of 7.)

```java
49  // indicate that one more buffer is occupied
50  ++occupiedBuffers;
51
52  // update writeLocation for future write operation
53  writeLocation = (writeLocation + 1) % buffers.length;
54
55  // display contents of shared buffers
56  System.err.println(createStateOutput());
57
58  notify(); // return a waiting thread to ready state
59  } // end method set
60
61  // return value from buffer
62  public synchronized int get()
63  {
64  // get name of thread that called this method
65  String name = Thread.currentThread().getName();
66
67  // while buffer is empty, place thread in waiting state
68  while (occupiedBuffers == 0)
69  {
```

© 2004 Deitel & Associates, Inc. All rights reserved.
6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 synchronizedBuffer controls access to a shared array of integers. (Part 5 of 7.)

```java
// occupiedBuffers:
--occupiedBuffers;

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

// display contents of shared buffers
System.err.println(createStateOutput());
notify();// return a waiting thread to ready state

return readValue;
} // end method get

// create state output
public String createStateOutput()
{
    // First line of state information
    String output = "buffers occupied: " +
    occupiedBuffers + "\nbuffers: ";

    for (int i = 0; i < buffers.length; ++i )
    {
        output += " " + buffers[ i ] + " ";
    } // end for

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

    for (int i = 0; i < buffers.length; ++i )
    {
        output += "---- ";
    } // end for

    // 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 ";
        } // end if
    }
}
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 synchronizedBuffer controls access to a shared array of integers. (Part 6 of 7.)
6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 synchronizedBuffer controls access to a shared array of integers. (Part 7 of 7.)

```java
else if (i == writeLocation)
{
  output += " W ";
}
else if (i == readLocation)
{
  output += " R ";
}
else
{
  output += " ";
}
```

Figure 6.7 CircularBufferTest instantiates producer and consumer threads. (Part 1 of 9.)

```java
// Fig. 6.7: CircularBufferTest.java
// CircularBufferTest shows two threads manipulating a
// circular buffer.

public class CircularBufferTest
{
  public static void main ( String args[] )
  {
    // create shared object for threads; use a reference
    // to a CircularBuffer rather than a Buffer reference
    // to invoke CircularBuffer method createStateOutput
    CircularBuffer sharedLocation = new CircularBuffer();

    // display initial state of buffers in CircularBuffer
    System.err.println( sharedLocation createStateOutput() );

    // set up threads
    Producer producer = new Producer( sharedLocation );
    Consumer consumer = new Consumer( sharedLocation );
  }
} // end class CircularBufferTest
```
6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 2 of 9.)

```java
22    producer.start(); // start producer thread
23    consumer.start(); // start consumer thread
24 }
25 // end main
26 } // end class CircularBufferTest
```

Sample Output:

<table>
<thead>
<tr>
<th>buffers occupied: 0</th>
<th>buffers: 11 11 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR</td>
<td>WR</td>
</tr>
</tbody>
</table>

All buffers empty. Consumer waits.

Producer writes 11
<table>
<thead>
<tr>
<th>buffers occupied: 1</th>
<th>buffers: 11 11 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>W</td>
</tr>
</tbody>
</table>

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 3 of 9.)

Consumer reads 11
<table>
<thead>
<tr>
<th>buffers occupied: 0</th>
<th>buffers: 11 11 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR</td>
<td>WR</td>
</tr>
</tbody>
</table>

Producer writes 12
<table>
<thead>
<tr>
<th>buffers occupied: 1</th>
<th>buffers: 11 12 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>W</td>
</tr>
</tbody>
</table>

Producer writes 13
<table>
<thead>
<tr>
<th>buffers occupied: 2</th>
<th>buffers: 11 12 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>R</td>
</tr>
</tbody>
</table>
6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 CircularBufferTest instantiates producer and consumer threads. (Part 4 of 9.)

```
Consumer reads 12
(buffers occupied: 1)
buffers: 11 12 13
                  W R
Producer writes 14
(buffers occupied: 2)
buffers: 14 12 13
                  W R
Producer writes 15
(buffers occupied: 3)
buffers: 14 15 13
                  WR
```

© 2004 Deitel & Associates, Inc. All rights reserved.
Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 6 of 9.)

Sample Output (Cont.):

Consumer reads 14
(buffers occupied: 2)
buffers: 14 15 16
   --- --- ---
   W   R

Producer writes 17
(buffers occupied: 3)
buffers: 17 15 16
   --- --- ---
   WR

Consumer reads 15
(buffers occupied: 2)
buffers: 17 15 16
   --- --- ---
   W   R

Figure 6.7 `CircularBufferTest` instantiates producer and consumer threads. (Part 7 of 9.)

Consumer reads 16
(buffers occupied: 1)
buffers: 17 15 16
   --- --- ---
   R   W

Consumer reads 17
(buffers occupied: 0)
buffers: 17 15 16
   --- --- ---
   WR

Producer writes 18
(buffers occupied: 1)
buffers: 17 18 16
   --- --- ---
   R   W
6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 CircularBufferTest instantiates producer and consumer threads. (Part 8 of 9.)

Sample Output (Cont.):

Consumer reads 18
(buffers occupied: 0)
buffers: 17 18 16

WR

All buffers empty. Consumer waits.

Producer writes 19
(buffers occupied: 1)
buffers: 17 18 19

W R

Producer done producing.
Terminating Producer.

Consumer reads 19
(buffers occupied: 0)
buffers: 17 18 19

WR

Consumer read values totaling: 155.
Terminating Consumer.

© 2004 Deitel & Associates, Inc. All rights reserved.