PRODUTTORE - CONSUMATORE CON SINCRONIZZAZIONE

Sfrutta le classi Producer, Consumer, Buffer dell'esempio non sincronizzato. Utilizza la nuova classe SynchronizedBuffer ed il nuovo main di SharedBufferTest_Synchronized.java

SharedBufferTest.java

```
public class SharedBufferTest_Synchronized
{
    public static void main( String [] args )
    {
        Buffer sharedLocation = new SynchronizedBuffer();

        Producer producer = new Producer( sharedLocation );
        Consumer consumer = new Consumer( sharedLocation );
        producer.start();  // start producer thread
        consumer.start();  // start consumer thread
    } // end main

} // end class SharedBufferTest
```

SynchronizedBuffer.java

```
public class SynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by Producer and Consumer
   private int occupiedBuffers = 0; // new-SYNCH

   // parola chiave synchronized
   // When one thread is executing a synchronized method for an object,
   // all other threads that invoke synchronized methods for the same
   // object block (suspend execution) until the first thread is done
   // with the object.
   // When a synchronized method exits, it automatically guarantees that
   // changes to the state of the object are visible to all threads.
   // General rule: if an object is visible to more than one thread, all
   // reads or writes to that object's variables are done through
   // synchronized methods

   public synchronized void set( int value )
   {
      String name = Thread.currentThread().getName();

      // new-SYNCH: controllo se il buffer è pieno e aspetto
      while (occupiedBuffers == 1 ) {
        // wait
        try {
          showMessage("-- " + name + " tenta di scrivere.");
          showMessage("-- Buffer pieno... " + name + " waits..");
          wait();
        } // se il thread viene interrotto durante il waiting...
        catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
```

```
      // se siamo qui il buffer è vuoto (0) e posso scrivere
      buffer = value;

      // adesso il buffer è pieno (1) e non posso inserire nuovi valori
      ++occupiedBuffers;

      showMessage(name + " ha scritto " + buffer + " --> notify()");
      notify();
   } // end method set, rilascia il lock


   // return value from buffer
   public synchronized int get()
   {
      // il nome del thread che ha richiesto il metodo get()
     String name = Thread.currentThread().getName();

     // new-SYNCH: controllo se il buffer è vuoto e aspetto
     while (occupiedBuffers == 0 ) {
       // wait
       try {
         showMessage("-- " + name + " tenta di leggere.");
         showMessage("-- Buffer vuoto... " + name +" waits..");
         wait();
       } // se il thread viene interrotto durante il waiting...
       catch (InterruptedException e) {
         e.printStackTrace();
       }
     }

     // se sono qui il Buffer contiene un valore
     // decremento così che il Producer possa scrivere
     --occupiedBuffers;

      showMessage(name + " ha letto " + buffer + " --> notify()");
      notify();
      return buffer;
   } // end method get

   // metodo non synchronized, invocato da dentro il monitor
   public void showMessage( String message ) {
     System.err.println( message );
   } // end method showMessage

} // end class UnsynchronizedBuffer
```

BUFFER CIRCOLARE CON SINCRONIZZAZIONE

Sfrutta le classi Producer, Consumer, Buffer dell'esempio non sincronizzato. Producer e Consumer sono state leggermente modificate per produrre 10 valori invece di 4.
Utilizza la nuova classe CircularBuffer ed il nuovo main di CircularBufferTest.java

CircularBufferTest.java

```
public class CircularBufferTest
{
   public static void main ( String args[] )
   {
      // create shared object for threads
      CircularBuffer sharedLocation = new CircularBuffer();

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

      // create producer and consumer objectss
      Producer producer = new Producer( sharedLocation );
      Consumer consumer = new Consumer( sharedLocation );

      producer.start();
      consumer.start();
   } // end main

} // end class CircularBufferTest
```

CircularBuffer.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();

      // while buffer full, place thread in waiting state
      while ( occupiedBuffers == buffers.length ) {
         // output thread and buffer information, then wait
         try {
```

```
            System.err.println( "\nAll buffers full. " + name + "
waits." );
            wait(); // wait until space is available
         } // end try

         // if waiting thread interrupted, print stack trace
         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 ] + " " );

      // indicate that one more buffer is occupied
      ++occupiedBuffers;

      // update writeLocation for future write operation
      writeLocation = ( writeLocation + 1 ) % buffers.length;

      // display contents of shared buffers
      System.err.println( createStateOutput() );

      notify(); // return a waiting thread to ready state
   } // end method set

   // return value from buffer
   public synchronized int get()
   {
      // get name of thread that called this method
      String name = Thread.currentThread().getName();

      // while buffer is empty, place thread in waiting state
      while ( occupiedBuffers == 0 ) {
         // output thread and buffer information, then wait
         try {
            System.err.println( "\nAll buffers empty. " + name + "
waits." );
            wait(); // wait until buffer contains new data
         } // end try

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

      } // end while

      // obtain value at current readLocation
      int readValue = buffers[ readLocation ];

      // output consumed value
      System.err.println( "\n" + name + " reads " + readValue + " " );
```

```
      // decrement occupied buffers value
      --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 ] + "  ";
      }

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

      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  ";
         } // end if
         else if ( i == writeLocation ) {
            output += " W   ";
         } // end if
         else if ( i == readLocation ) {
            output += "  R  ";
         } // end if
         else {
            output += "     ";
         } // end else
      } // end for

      output += "\n";

      return output;
   } // end method createStateOutput
} // end class CircularBuffer
```