Multithreading
History of Multi-processing & Multithreading

- 60’s - I/O concurrency – (Operating Systems)
- Multi-user systems
- Pipelined/Vector processors
- 70’s - Co-processors
- 80’s - Parallel Processing Systems
  - SIMD, MIMD, shared/distributed memory
- Distributed Computing
- Threads, Dataflow Computing
- 90’s - Grid Computing
- 00’s – Multi-scale computing
23.2 Thread States: Life Cycle of a Thread
23.3 Thread Priorities and Thread Scheduling
23.4 Creating and Executing Threads
23.5 Thread Synchronization
23.6 Producer/Consumer Relationship without Synchronization
23.7 Producer/Consumer Relationship with Synchronization
23.8 Producer/Consumer Relationship: Circular Buffer
23.9 Producer/Consumer Relationship: ArrayBlockingQueue
23.10 Multithreading with GUI
23.11 Other Classes and Interfaces in java.util.concurrent
23.12 Monitors and Monitor Locks
23.13 Wrap-Up
23.1 Introduction: Multi-threading in Java

- Provides application with multiple threads of execution
- Allows programs to perform tasks concurrently
- Often requires programmer to synchronize threads to function correctly
A problem with single-threaded applications is that lengthy activities must complete before other activities can begin. In a multithreaded application, threads can be distributed across multiple processors (if they are available) so that multiple tasks are performed concurrently (simultaneously) and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems that simulate concurrency—when one thread cannot proceed, another can use the processor.
Portability Tip 23.1

Unlike languages that do not have built-in multithreading capabilities (such as C and C++) and must therefore make nonportable calls to operating system multithreading primitives, Java includes multithreading primitives as part of the language itself and as part of its libraries. This facilitates manipulating threads in a portable manner across platforms.
23.2 Thread States: Life Cycle of a Thread

• Thread states
  – new state
    • New thread begins its life cycle in the new state
    • Remains in this state until program starts the thread, placing it in the runnable state
  – runnable state
    • A thread in this state is executing its task
  – waiting state
    • A thread transitions to this state to wait for another thread to perform a task
23.2 Thread States: Life Cycle of a Thread

• Thread states
  – *timed waiting* state
    • A thread enters this state to wait for another thread or for an amount of time to elapse
    • A thread in this state returns to the *runnable* state when it is signaled by another thread or when the timed interval expires
  – *terminated* state
    • A *runnable* thread enters this state when it completes its task
Fig. 23.1 | Thread life-cycle UML state diagram.
23.2 Thread States: Life Cycle of a Thread

- Operating system view of *runnable* state
  - *ready* state
    - A thread in this state is not waiting for another thread, but is waiting for the operating system to assign the thread a processor
  - *running* state
    - A thread in this state currently has a processor and is executing
    - A thread in the *running* state often executes for a small amount of processor time called a time slice or quantum before transitioning back to the *ready* state
Fig. 23.2 | Operating system’s internal view of Java’s runnable state.
23.3 Thread Priorities and Thread Scheduling

• Priorities
  – Every Java thread has a priority
  – Java priorities are in the range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10)
  – Threads with a higher priority are more important and will be allocated a processor before threads with a lower priority
  – Default priority is `NORM_PRIORITY` (a constant of 5)
23.3 Thread Priorities and Thread Scheduling

• Thread scheduler
  – Determine which thread runs next
  – Simple implementation runs equal-priority threads in a round-robin fashion
  – Higher-priority threads can preempt the currently running thread
  – In some cases, higher-priority threads can indefinitely postpone lower-priority threads which is also known as starvation
Portability Tip 23.2

Thread scheduling is platform dependent—an application that uses multithreading could behave differently on separate Java implementations.
Portability Tip 23.3

When designing applets and applications that use threads, you must consider the threading capabilities of all the platforms on which the applets and applications will execute.
Fig. 23.3 | Thread-priority scheduling.
23.4 Creating and Executing Threads

- **Runnable interface**
  - Preferred means of creating a multithreaded application
  - Declares method `run`
  - Executed by an object that implements the `Executor` interface

- **Executor interface**
  - Declares method `execute`
  - Creates and manages a group of threads called a thread pool
23.4 Creating and Executing Threads

• **ExecutorService** interface
  – Subinterface of **Executor** that declares other methods for managing the life cycle of an **Executor**
  – Can be created using static methods of class **Executors**
  – Method **shutdown** ends threads when tasks are completed

• **Executors** class
  – Method **newFixedThreadPool** creates a pool consisting of a fixed number of threads
  – Method **newCachedThreadPool** creates a pool that creates new threads as they are needed
class PrintTask implements Runnable {
    private int sleepTime; // random sleep time for thread
    private String threadName; // name of thread
    private static Random generator = new Random();

    // assign name to thread
    public PrintTask( String name )
    {
        threadName = name; // set name of thread

        // pick random sleep time between 0 and 5 seconds
        sleepTime = generator.nextInt( 5000 );
    } // end PrintTask constructor

    // implement Runnable to create separate thread
}
// method run is the code to be executed by new thread
public void run() {
    try // put thread to sleep for sleepTime amount of time
    {
        System.out.printf( "%s going to sleep for %d milliseconds.
", threadName, sleepTime );
        Thread.sleep( sleepTime ); // put thread to sleep
    } // end try
    // if thread interrupted while sleeping, print stack trace
    catch ( InterruptedException exception )
    {
        exception.printStackTrace();
    } // end catch
    // print thread name
    System.out.printf( "%s done sleeping\n", threadName );
} // end method run
} // end class PrintTask

Declare run method to fulfill interface
// Fig. 23.5: RunnableTester.java
// Multiple threads printing at different intervals.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester
{
    public static void main(String[] args)
    {
        // create and name each runnable
        PrintTask task1 = new PrintTask("thread1");
        PrintTask task2 = new PrintTask("thread2");
        PrintTask task3 = new PrintTask("thread3");

        System.out.println("Starting threads");

        // create ExecutorService to manage threads
        ExecutorService threadExecutor = Executors.newFixedThreadPool(3);

        // start threads and place in runnable state
        threadExecutor.execute(task1); // start task1
        threadExecutor.execute(task2); // start task2
        threadExecutor.execute(task3); // start task3

        threadExecutor.shutdown(); // shutdown worker threads
    }
}
RunnableTester.java

(2 of 2)
23.5.1 Array Writer (unsynchronized)

- Dangerous situation of uncoordinated multiple writers to a shared data structure.
  - Each of 2 threads writes 3 values
  - Each thread writes to “successive” locations
  - Each thread makes a copy of the array index then sleeps a while before using that index copy to write to the array
  - The sleep simply exaggerates an underlying problem
// Fig. 23.7: SimpleArray.java
// Class that manages an integer array to be shared by multiple threads.
import java.util.Random;
public class SimpleArray // CAUTION: NOT THREAD SAFE!
{
    private final int array[]; // the shared integer array
    private int writeIndex = 0; // index of next element to write
    private final static Random generator = new Random();

    // construct a SimpleArray of a given size
    public SimpleArray( int size )
    {
        array = new int[ size ];
    } // end constructor

} // end class SimpleArray
public void add(int value) {
    int position = writeIndex; // store the write index

    try {
        // put thread to sleep for 0-499 milliseconds
        Thread.sleep(generator.nextInt(500));
    } // end try
    catch (InterruptedException ex) {
        ex.printStackTrace();
    } // end catch

    // put value in the appropriate element
    array[position] = value;
    System.out.printf("%s wrote %2d to element %d.\n", Thread.currentThread().getName(), value, position);

    ++writeIndex; // increment index of element to be written next
    System.out.printf("Next write index: %d\n", writeIndex);
} // end method add
// used for outputting the contents of the shared integer array
public String toString()
{
    String arrayString = "\nContents of SimpleArray: \n";

    for (int i = 0; i < array.length; i++)
        arrayString += array[i] + " ";

    return arrayString;
}

} // end method toString

} // end class SimpleArray
// Fig. 23.8: ArrayWriter.java
// Adds integers to an array shared with other Runnables
import java.lang.Runnable;

public class ArrayWriter implements Runnable
{
    private final SimpleArray sharedSimpleArray;
    private final int startValue;

    public ArrayWriter( int value, SimpleArray array )
    {
        startValue = value;
        sharedSimpleArray = array;
    } // end constructor

    public void run()
    {
        for ( int i = startValue; i < startValue + 3; i++ )
        {
            sharedSimpleArray.add( i ); // add an element to the shared array
        } // end for
    } // end method run
} // end class ArrayWriter
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class SharedArrayTest
{
    public static void main(String[] arg)
    {
        // construct the shared object
        SimpleArray sharedSimpleArray = new SimpleArray(6);

        // create two tasks to write to the shared SimpleArray
        ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
        ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);

        // execute the tasks with an ExecutorService
        ExecutorService executor = Executors.newCachedThreadPool();
        executor.execute(writer1);
        executor.execute(writer2);
        executor.shutdown();
    }
}
try
{
    // wait 1 minute for both writers to finish executing
    boolean tasksEnded = executor.awaitTermination(1, TimeUnit.MINUTES);

    if (tasksEnded)
        System.out.println(sharedSimpleArray); // print contents
    else
        System.out.println("Timed out while waiting for tasks to finish.");
} // end try
catch (InterruptedException ex)
{
    System.out.println("Interrupted while wait for tasks to finish.");
} // end catch
} // end main
} // end class SharedArrayTest
pool-1-thread-1 wrote 1 to element 0
Next write index: 1
pool-1-thread-2 wrote 11 to element 0
Next write index: 2
pool-1-thread-2 wrote 12 to element 2
Next write index: 3
pool-1-thread-1 wrote 2 to element 1
Next write index: 4
pool-1-thread-1 wrote 3 to element 4
Next write index: 5
pool-1-thread-2 wrote 13 to element 3
Next write index: 6

Contents of SimpleArray:
11 2 12 13 3 0

pool-1-thread-1 wrote 1 to element 0
Next write index: 1
pool-1-thread-1 wrote 2 to element 1
Next write index: 2
pool-1-thread-2 wrote 11 to element 0
Next write index: 3
pool-1-thread-1 wrote 3 to element 2
Next write index: 4
pool-1-thread-2 wrote 12 to element 3
Next write index: 5
pool-1-thread-2 wrote 13 to element 5
Next write index: 6

Contents of SimpleArray:
11 2 3 12 0 13
23.5.1 Array Writer (synchronized)

• Problems with previous version
  – Recognize that the fundamental problem with previous version is that the “add” method must be executed atomically
  – Once a thread enters the “add” method, it must not be interrupted by another thread, and must be allowed to complete
  – No other thread can “enter” the add method while another thread has partially executed its add method call.
import java.util.Random;

public class SimpleArray {
    private final int array[]; // the shared integer array
    private int writeIndex = 0; // index of next element to be written
    private final static Random generator = new Random();

    // construct a SimpleArray of a given size
    public SimpleArray( int size )
    {
        array = new int[ size ];
    } // end constructor
public synchronized void add(int value) {
    int position = writeIndex; // store the write index
    try {
        // put thread to sleep for 0-499 milliseconds
        Thread.sleep(generator.nextInt(500));
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    // put value in the appropriate element
    array[position] = value;
    System.out.printf("%s wrote %d to element %d.\n", Thread.currentThread().getName(), value, position);
    ++writeIndex; // increment index of element to be written next
    System.out.printf("Next write index: %d\n", writeIndex);
} // end method add

public String toString() {
    String arrayString = "Contents of SimpleArray:\n";
    for (int i = 0; i < array.length; i++)
        arrayString += array[i] + " ";
    return arrayString;
} // end method toString
} // end class SimpleArray
23.6 Producer/Consumer Relationship without Synchronization

• **Producer/consumer relationship**
  – Producer generates data and stores it in shared memory
  – Consumer reads data from shared memory
  – Shared memory is called the buffer
Fig. 23.6 | Buffer interface used in producer/consumer examples.
public class Producer implements Runnable {
  private static Random generator = new Random();
  private Buffer sharedLocation; // reference to shared object

  // constructor
  public Producer(Buffer shared) {
    sharedLocation = shared;
  } // end Producer constructor

  // store values from 1 to 10 in sharedLocation
  public void run() {
    int sum = 0;
    ...
```java
for ( int count = 1; count <= 10; count++ )
{
    try // sleep 0 to 3 seconds, then place value in Buffer
    {
        Thread.sleep( generator.nextInt( 3000 ) ); // sleep thread
        sharedLocation.set( count ); // set value in buffer
        sum += count; // increment sum of values
        System.out.printf( "%2d\n", sum );
    } // end try
    // if sleeping thread interrupted, print stack trace
    catch ( InterruptedException exception )
    {
        exception.printStackTrace();
    } // end catch
} // end for

System.out.printf( "\n%\n%\n%\n%\”, "Producer done producing.", "Terminating Producer." );
} // end method run
} // end class Producer
```

Sleep for up to 3 seconds
// Fig. 23.8: Consumer.java
// Consumer's run method loops ten times reading a value from buffer.
import java.util.Random;

public class Consumer implements Runnable
{
    private static Random generator = new Random();
    private Buffer sharedLocation; // reference to shared object

    // constructor
    public Consumer( Buffer shared )
    {
        sharedLocation = shared;
    } // end Consumer constructor

    // read sharedLocation's value four times and sum it
    public void run()
    {
        int sum = 0;

        Implement the Runnable interface so that producer will run in a separate thread

        Declare run method to satisfy interface
    } // end run method
for ( int count = 1; count <= 10; count++ )
{
    // sleep 0 to 3 seconds, read value from buffer and add to sum
    try
    {
        Thread.sleep( generator.nextInt( 3000 ) );
        sum += sharedLocation.get();
        System.out.printf( "\tt\t%2d\n", sum );
    } // end try
    // if sleeping thread interrupted, print stack trace
    catch ( InterruptedException exception )
    {
        exception.printStackTrace();
    } // end catch
} // end for
System.out.printf( "\n%2d.%n", sum, "Terminating Consumer." );
} // end method run
} // end class Consumer
public class UnsynchronizedBuffer implements Buffer {
    private int buffer = -1; // shared by producer and consumer threads

    // place value into buffer
    public void set(int value) {
        System.out.printf( "Producer writes\t%2d", value );
        buffer = value;
    } // end method set

    // return value from buffer
    public int get() {
        System.out.printf( "Consumer reads\t%2d", buffer );
        return buffer;
    } // end method get
} // end class UnsynchronizedBuffer
// Fig 23.10: SharedBufferTest.java
// Application shows two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest
{
    public static void main( String[] args )
    {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );

        // create UnsynchronizedBuffer to store ints
        Buffer sharedLocation = new UnsynchronizedBuffer();
    }
}
System.out.println( "Action		Value	Produced	Consumed" );
System.out.println( "------		-----	--------	--------" );

// try to start producer and consumer giving each of them access
// to sharedLocation
try {
    application.execute( new Producer( sharedLocation ) );
    application.execute( new Consumer( sharedLocation ) );
} // end try
catch ( Exception exception ) {
    exception.printStackTrace();
} // end catch

application.shutdown(); // terminate application when threads end
} // end main
} // end class SharedBufferTest

Pass shared buffer to both producer and consumer
<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
<th>Produced</th>
<th>Consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer writes</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Producer writes</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>4</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Producer writes</td>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>7</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>7</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>Consumer reads</td>
<td>7</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>Producer writes</td>
<td>8</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>8</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>Producer writes</td>
<td>8</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>9</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>10</td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

Producer done producing.
Terminating Producer.

Consumer reads 10
Consumer reads 10
Consumer reads 10
Consumer reads 10

Consumer read values totaling 77.
Terminating Consumer.
<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
<th>Produced</th>
<th>Consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer reads</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>3</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>4</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Producer writes</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>6</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

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.
23.6 Producer/Consumer Relationship with Synchronization (spin-locks; busy-waiting)

- Add a boolean variable (bufferFull) and method (isFull) to the Buffer implementation
- Use a busy-wait while loop in the run methods of the Producer and the Consumer to keep from overwriting an unread value and from re-reading a stale value from the buffer.
Buffer.java

Fig. 23.6 | Buffer interface used in producer/consumer examples.
// Fig. 23.7: Producer.java
// Producer's run method stores the values 1 to 10 in buffer.
import java.util.Random;

public class Producer implements Runnable
{

    private static Random generator = new Random();
    private Buffer sharedLocation; // reference to shared object

    // constructor
    public Producer( Buffer shared )
    {
        sharedLocation = shared;
    } // end Producer constructor

    // store values from 1 to 10 in sharedLocation
    public void run()
    {
        int sum = 0;
    }
} // end class Producer
for ( int count = 1; count <= 10; count++ )
{
    try // sleep 0 to 3 seconds, then place value in Buffer
    {
        Thread.sleep( generator.nextInt( 3000 ) ); // sleep thread
        while (sharedLocation.isFull()); // do nothing
        sharedLocation.set( count ); // set value in buffer
        sum += count; // increment sum of values
        System.out.printf("	%2d
", sum);
    } // end try
    // if sleeping thread interrupted, print stack trace
    catch ( InterruptedException exception )
    {
        exception.printStackTrace();
    } // end catch
} // end for

System.out.printf( "\nProducer done producing.\n" , "Terminating Producer."");
} // end method run
} // end class Producer
public class Consumer implements Runnable {
    private static Random generator = new Random();
    private Buffer sharedLocation; // reference to shared object

    // constructor
    public Consumer( Buffer shared ) {
        sharedLocation = shared;
    } // end Consumer constructor

    // read sharedLocation's value four times and sum it
    public void run() {
        int sum = 0;
    }
} // end Consumer class
for ( int count = 1; count <= 10; count++ )
{
    // sleep 0 to 3 seconds, read value from buffer and add to sum
    try
    {
        Thread.sleep( generator.nextInt( 3000 ) );
        // do nothing
        sum += sharedLocation.get();
        System.out.printf( "\t\t%2d
", sum );
    } // end try
    // if sleeping thread interrupted, print stack trace
    catch ( InterruptedException exception )
    {
        exception.printStackTrace();
    } // end catch
} // end for

System.out.printf( "\n%6d. \n%6s
", 
"Consumer read values totaling", sum, "Terminating Consumer."
);
public class UnsynchronizedBuffer implements Buffer {
    private int buffer = -1; // shared by producer and consumer threads
    private int bufferFull = false; // flag to indicate when buffer has an unread value

    // place value into buffer
    public void set(int value) {
        System.out.printf("Producer writes\t%2d", value);
        buffer = value;
        bufferFull = true;
    } // end method set

    // return value from buffer
    public int get() {
        System.out.printf("Consumer reads\t%2d", buffer);
        bufferFull = false; // note an error could still occur in this example if the thread is interrupted at this point
        return buffer;
    } // end method get
} // end class UnsynchronizedBuffer
// Fig 23.10: SharedBufferTest.java
// Application shows two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest
{
    public static void main( String[] args )
    {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool(2);

        // create UnsynchronizedBuffer to store ints
        Buffer sharedLocation = new UnsynchronizedBuffer();
System.out.println( "Action		Value	Produced	Consumed" );
System.out.println( "------		-----	--------	--------" );

// try to start producer and consumer giving each of them access
// to sharedLocation
try
{
    application.execute( new Producer( sharedLocation ) );
    application.execute( new Consumer( sharedLocation ) );
} // end try
catch ( Exception exception )
{
    exception.printStackTrace();
} // end catch

application.shutdown(); // terminate application when threads end
} // end main
} // end class SharedBufferTest

Pass shared buffer to both producer and consumer
<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
<th>Sum of Produced</th>
<th>Sum of Consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer writes</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Producer writes</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Producer writes</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>3</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Producer writes</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>4</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Producer writes</td>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>5</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Producer writes</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>6</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>Producer writes</td>
<td>7</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>7</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>Producer writes</td>
<td>8</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>8</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>Producer writes</td>
<td>9</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>9</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>Producer writes</td>
<td>10</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>Producer done producing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminating Producer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer reads</td>
<td>10</td>
<td></td>
<td>55</td>
</tr>
</tbody>
</table>

Consumer read values totaling 55
Terminating Consumer
Problems with Busy-waiting

1. An error can still occur in the get code if the schedule switched threads after the flag changes but before the value is safely returned. In effect we reduced the possibility of error greatly, but it still is not entirely safe

2. The solution is inefficient: It would be much better to allow the thread to “give-up” the CPU when it is unable to proceed – instead of busy-waiting.
23.8 Monitors and Monitor Locks

• Monitors
  – Every object in Java has a monitor
  – Allows one thread at a time to execute inside a synchronized statement
  – Threads waiting to acquire the monitor lock are placed in the blocked state
  – Object method wait places a thread in the waiting state
  – Object method notify wakes up a waiting thread
  – Object method notifyAll wakes up all waiting threads
public class SynchronizedBuffer implements Buffer {
    private int buffer = -1; // shared by producer and consumer threads
    private boolean occupied = false; // count of occupied buffers

    // place value into buffer
    public synchronized void set(int value) {
        // while there are no empty locations, place thread in waiting state
        while (occupied)
        {
            // output thread information and buffer information, then wait
            try {
                System.out.println("Producer tries to write.");
                displayState("Buffer full. Producer waits.");
                wait();
            } // end try
            catch (InterruptedException exception)
            {
                exception.printStackTrace();
            } // end catch
        } // end while
        buffer=value; // set new buffer value
    }
}
Buffer is now occupied

Notify waiting thread that it may now read a value

Declare synchronized method

Wait until buffer is full

SynchronizedBuffer

•
Buffer is now empty

Notify thread it may now write to buffer

SynchronizedBuffer

(3 of 3)
```java
// Fig 23.20: SharedBufferTest2.java
// Application shows two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest2
{
    public static void main( String[] args )
    {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );

        // create SynchronizedBuffer to store ints
        Buffer sharedLocation = new SynchronizedBuffer();

        System.out.printf( "%-40s%-40s%-40s
", "Operation", "Buffer", "Occupied" );

        try // try to start producer and consumer
        {
            application.execute( new Producer( sharedLocation ) );
            application.execute( new Consumer( sharedLocation ) );
        } // end try
        catch ( Exception exception )
        {
            exception.printStackTrace();
        } // end catch
    }
}
```

- Create `SynchronizedBuffer` for use in producer and consumer.
- Execute the producer and consumer in separate threads.
```java
application.shutdown();
}
} // end class SharedBufferTest2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer tries to read.</td>
<td>-1</td>
<td>false</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 5</td>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 5</td>
<td>5</td>
<td>false</td>
</tr>
</tbody>
</table>
```
### Synchronize dBBuffer.java

<table>
<thead>
<tr>
<th>Event</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer writes 6</td>
<td>6</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 6</td>
<td>6</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td>6</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 7</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 7</td>
<td>7</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td>7</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 8</td>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 8</td>
<td>8</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 9</td>
<td>9</td>
<td>true</td>
</tr>
<tr>
<td>Producer tries to write.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer full. Producer waits.</td>
<td>9</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 9</td>
<td>9</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 10</td>
<td>10</td>
<td>true</td>
</tr>
</tbody>
</table>

Producer done producing.
Terminating Producer.
Consumer reads 10
Consumer read values totaling 55.
Terminating Consumer.

© 2005 Pearson Education, Inc. All rights reserved.
23.10 Thread Synchronization – Locks and Conditions

• The Lock/Condition interface mechanism was new with Java 5. Lock interface
  – lock method obtains the lock, enforcing mutual exclusion
  – unlock method releases the lock
  – Class ReentrantLock implements the Lock interface

• This is a finer-grained resolution of locking than synchronizing an entire method.

• But… usually considered more clumsy than Wait/Notify. Harder to debug, but potentially provides better performance than monitors.
// Fig. 23.11: SynchronizedBuffer.java
// SynchronizedBuffer synchronizes access to a single shared integer.
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class SynchronizedBuffer implements Buffer {
    // Lock to control synchronization with this buffer
    private Lock accessLock = new ReentrantLock();

    // conditions to control reading and writing
    private Condition canWrite = accessLock.newCondition();
    private Condition canRead = accessLock.newCondition();

    private int buffer = -1; // shared by producer and consumer threads
    private boolean occupied = false; // whether buffer is occupied

    // place int value into buffer
    public void set(int value) {
        accessLock.lock(); // lock this object
        try {
            // Try to obtain the lock before setting the value of the shared data
        } finally {
            accessLock.unlock(); // unlock this object
        }
    }
}
// output thread information and buffer information, then wait
try {
    // while buffer is not empty, place thread in waiting state
    while (occupied) {
        System.out.println("Producer tries to write.");
        displayState("Buffer full. Producer waits.");
        canWrite.await(); // wait until buffer is empty
    } // end while

    buffer = value; // set new buffer value

    // indicate producer cannot store another value
    // until consumer retrieves current buffer value
    occupied = true;
displayState("Producer writes " + buffer);

// signal thread waiting to read from buffer
canRead.signal();
} // end try
catch (InterruptedException exception)
{
    exception.printStackTrace();
} // end catch
finally
{
    accessLock.unlock(); // unlock this object
} // end finally
} // end method set

// return value from buffer
public int get()
{
    int readValue = 0; // initialize value read from buffer
    accessLock.lock(); // lock this object
// output thread information and buffer information, then wait
try
{
    // while no data to read, place thread in waiting state
    while ( !occupied )
    {
        System.out.println( "Consumer tries to read." );
displayState( "Buffer empty. Consumer waits." );
canRead.await(); // wait until buffer is full
    } // end while

    // indicate that producer can store another value
    // because consumer just retrieved buffer value
    occupied = false;

    readValue = buffer; // retrieve value from buffer
displayState( "Consumer reads " + readValue );

} // end try
// signal thread waiting for buffer to be empty
    canWrite.signal();
} // end try

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

finally
{
    accessLock.unlock(); // unlock this object
} // end finally

    return readValue;
} // end method get

// display current operation and buffer state
public void displayState(String operation)
{
    System.out.printf("%40s\t\t%d\t%bn", operation, buffer,
    occupied);
} // end method displayState

} // end class SynchronizedBuffer

// Signal producer that it can write to buffer
// Release lock on shared data
Common Programming Error 23.3

Place calls to `Lock` method `unlock` in a `finally` block. If an exception is thrown, `unlock` must still be called or deadlock could occur.
Software Engineering Observation 23.2

Always invoke method `await` in a loop that tests an appropriate condition. It is possible that a thread will reenter the `runnable` state before the condition it was waiting on is satisfied. Testing the condition again ensures that the thread will not erroneously execute if it was signaled early.
Common Programming Error 23.4

Forgetting to signal a thread that is waiting for a condition is a logic error. The thread will remain in the waiting state, which will prevent the thread from doing any further work. Such waiting can lead to indefinite postponement or deadlock.
// Fig 23.12: SharedBufferTest2.java
// Application shows two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest2
{
    public static void main(String[] args)
    {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool(2);

        // create SynchronizedBuffer to store ints
        Buffer sharedLocation = new SynchronizedBuffer();
    }
}
System.out.printf("%-40s%s	%-40s%s
", "Operation", "Buffer", "Occupied", "---------");

try // try to start producer and consumer
{
    application.execute(new Producer(sharedLocation));
    application.execute(new Consumer(sharedLocation));
} // end try

catch (Exception exception)
{
    exception.printStackTrace();
} // end catch

application.shutdown();

} // end main

} // end class SharedBufferTest2

Execute the producer and consumer in separate threads
<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>Producer tries to write.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer full. Producer waits.</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>Producer tries to write.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer full. Producer waits.</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 5</td>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 5</td>
<td>5</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td>5</td>
<td>false</td>
</tr>
<tr>
<td>Event</td>
<td>Value</td>
<td>In buffer</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------</td>
<td>-----------</td>
</tr>
<tr>
<td>Producer writes 6</td>
<td>6</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 6</td>
<td>6</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 7</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 7</td>
<td>7</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 8</td>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 8</td>
<td>8</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 9</td>
<td>9</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 9</td>
<td>9</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 10</td>
<td>10</td>
<td>true</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 10</td>
<td>10</td>
<td>false</td>
</tr>
</tbody>
</table>

Consumer read values totaling 55.  
Terminating Consumer.
23.8 Producer/Consumer Relationship

Circular Buffer

• Circular buffer
  – Provides extra buffer space into which producer can place values and consumer can read values
Performance Tip 23.4

Even when using a circular buffer, it is possible that a producer thread could fill the buffer, which would force the producer thread to wait until a consumer consumes a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, the consumer thread must wait until the producer produces another value. The key to using a circular buffer is to optimize the buffer size to minimize the amount of thread wait time.
// Fig. 23.13: CircularBuffer.java

// SynchronizedBuffer synchronizes access to a single shared integer.
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

// Lock to control synchronization with this buffer
private Lock accessLock = new ReentrantLock();

// conditions to control reading and writing
private Condition canWrite = accessLock.newCondition();
private Condition canRead = accessLock.newCondition();

private int[] buffer = {-1, -1, -1};

private int occupiedBuffers = 0; // count number of buffers used
private int writeIndex = 0; // index to write next value
private int readIndex = 0; // index to read next value

// place value into buffer
public void set(int value) {
    accessLock.lock(); // lock this object
    // Lock to impose mutual exclusion
    // Condition variables to control writing and reading
    // Circular buffer; provides three spaces for data
    // Obtain the lock before writing data to the circular buffer
}
// output thread information and buffer information, then wait
try {
  // while no empty locations, place thread in waiting state
  while (occupiedBuffers == buffer.length) {
    System.out.printf("All buffers full. Producer waits.\n");
    canWrite.await(); // await until a buffer element is free
  } // end while

  buffer[writeIndex] = value; // set new buffer
  // update circular write index
  writeIndex = (writeIndex + 1) % buffer.length;
  occupiedBuffers++; // one more buffer element
  displayState("Producer writes " + buffer[writeIndex]);
  canRead.signal(); // signal threads waiting to read from buffer
} // end try
catch (InterruptedException exception) {
} // end catch
finally {
  accessLock.unlock(); // unlock this object
} // end finally
} // end method set
public int get()
{
    int readValue = 0; // initialize value read from buffer
    accessLock.lock(); // lock this object

    // wait until buffer has data, then read value
    try
    {
        // while no data to read, place thread in waiting state
        while (occupiedBuffers == 0)
        {
            System.out.printf( "All buffers empty. Consumer waits.\n" );
            canRead.await(); // await until a buffer element is filled
        } // end while

        readValue = buffer[readIndex]; // read value

        // update circular read index
        readIndex = (readIndex + 1) % buffer.length;
    }
    catch (InterruptedException e)
    {
        // do nothing
    }
    finally
    {
        accessLock.unlock(); // unlock object
    }

    return readValue; // return value from buffer
}
occupiedBuffers--; // one more buffer element is empty

displayState( "Consumer reads " + readValue );

canWrite.signal(); // signal threads waiting to write to buffer

} // end try

// if waiting thread interrupted, print stack trace

catch ( InterruptedException exception )
{
    exception.printStackTrace();
} // end catch

finally
{
    accessLock.unlock(); // unlock this object
} // end finally

return readValue;
} // end method get

// display current operation and buffer state

public void displayState( String operation )
{
    // output operation and number of occupied buffers
    System.out.printf( "%s
", operation, " (buffers occupied: ", occupiedBuffers, "buffers: ");

    for ( int value : buffer )
        System.out.printf( " %2d ", value ); // output values in buffer
System.out.println("\n");

for (int i = 0; i < buffer.length; i++)
    System.out.print("---- ");

System.out.println("\n");

for (int i = 0; i < buffer.length; i++)
{
    if (i == writeIndex && i == readIndex)
        System.out.print(" WR"); // both write and read index
    else if (i == writeIndex)
        System.out.print(" W   "); // just write index
    else if (i == readIndex)
        System.out.print("  R  "); // just read index
    else
        System.out.print("     "); // neither index
}

System.out.println("\n"); // end method displayState

System.out.println("\n"); // end class CircularBuffer

Outline

CircularBuffer.java

(5 of 5)
// Fig 23.14: CircularBufferTest.java
// Application shows two threads manipulating a circular buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CircularBufferTest
{
    public static void main( String[] args )
    {
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );
        // create CircularBuffer to store ints
        Buffer sharedLocation = new CircularBuffer();
        try // try to start producer and consumer
        {
            application.execute( new Producer( sharedLocation ) );
            application.execute( new Consumer( sharedLocation ) );
        } // end try
        catch ( Exception exception )
        {
            exception.printStackTrace();
        } // end catch
        application.shutdown();
    } // end main
} // end class CircularBufferTest
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
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

© 2005 Pearson Education, Inc. All rights reserved.
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.
23.10 Multithreading with GUI

• Swing GUI components
  – Not thread safe
  – Updates should be performed in the event-dispatching thread
  
  • Use static method `invokeLater` of class `SwingUtilities` and pass it a `Runnable` object
// Fig. 23.17: RunnableObject.java
// Runnable that writes a random character to a JLabel
import java.util.Random;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
import java.awt.Color;

public class RunnableObject implements Runnable {
    private static Random generator = new Random(); // for random letters
    private Lock lockObject; // application lock; passed in to constructor
    private Condition suspend; // used to suspend and resume thread
    private boolean suspended = false; // true if thread suspended
    private JLabel output; // JLabel for output

    public RunnableObject( Lock theLock, JLabel label ) {
        lockObject = theLock; // store the Lock for the application
        suspend = lockObject.newCondition(); // create new Condition
        output = label; // store JLabel for outputting character
    }

    public void run() {
        // get name of executing thread
        final String threadName = Thread.currentThread().getName();
    }
}
while (true) // infinite loop; will be terminated from outside
{
    try
    {
        // sleep for up to 1 second
        Thread.sleep(generator.nextInt(1000));

        lockObject.lock(); // obtain the lock
        try
        {
            while (suspended) // loop until not suspended
            {
                suspend.await(); // suspend thread execution
            } // end while
        } // end try
        finally
        {
            lockObject.unlock(); // unlock the lock
        } // end finally
    } // end try
    // if thread interrupted during wait/sleep
    catch (InterruptedException exception)
    {
        exception.printStackTrace(); // print stack trace
    } // end catch
RunnableObject.java

57 // display character on corresponding JLabel
58 SwingUtilities.invokeLater(
59     new Runnable()
60     {
61         // pick random character and display it
62         public void run()
63         {
64             // select random uppercase letter
65             char displayChar =
66             ( char ) ( generator.nextInt( 26 ) + 65 );
67
68             // output character in JLabel
69             output.setText( threadName + ": " + displayChar );
70         } // end method run
71     } // end inner class
72 ); // end call to SwingUtilities.invokeLater
73 } // end while
74 } // end method run

© 2005 Pearson Education, Inc. All rights reserved.
// change the suspended/running state
public void toggle()
{
    suspended = !suspended; // toggle boolean controlling state

    // change label color on suspend/resume
    output.setBackground( suspended ? Color.RED : Color.GREEN );

    lockObject.lock(); // obtain lock
    try
    {
        if ( !suspended ) // if thread resumed
        {
            suspend.signal(); // resume thread
        } // end if
    } // end try
    finally
    {
        lockObject.unlock(); // release lock
    } // end finally
} // end method toggle
} // end class RunnableObject
// Fig. 23.18: RandomCharacters.java
// Class RandomCharacters demonstrates the Runnable interface
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class RandomCharacters extends JFrame implements ActionListener {
    private final static int SIZE = 3; // number of threads
    private JCheckBox checkboxes[]; // array of JCheckBoxes
    private Lock lockObject = new ReentrantLock( true ); // single lock

    // array of RunnableObjects to display random characters
    private RunnableObject[] randomCharacters = new RunnableObject[ SIZE ];

    Create Lock for the application
// set up GUI and arrays
public RandomCharacters()
{
    checkboxes = new JCheckBox[SIZE]; // allocate space for array
    setLayout(new GridLayout(SIZE, 2, 5, 5)); // set layout

    ExecutorService runner = Executors.newFixedThreadPool(SIZE);

    // loop SIZE times
    for (int count = 0; count < SIZE; count++)
    {
        JLabel outputJLabel = new JLabel(); // create JLabel
        outputJLabel.setBackground(Color.GREEN); // set color
        outputJLabel.setOpaque(true); // set JLabel to be opaque
        add(outputJLabel); // add JLabel to JFrame

        // create JCheckBox to control suspend/resume state
        checkboxes[count] = new JCheckBox("Suspended");

        // add listener which executes when JCheckBox is clicked
        checkboxes[count].addActionListener(this);
        add(checkboxes[count]); // add JCheckBox to JFrame
    }
}
// create a new RunnableObject
randomCharacters[count] = new RunnableObject(lockObject, outputJLabel);

// execute RunnableObject
runner.execute(randomCharacters[count]);
}

setSize(275, 90); // set size of window
setVisible(true); // show window

runner.shutdown(); // shutdown runner when threads finish
}

// handle JCheckBox events
public void actionPerformed(ActionEvent event)
{
    // loop over all JCheckBoxes in array
    for (int count = 0; count < checkboxes.length; count++)
    {
        // check if this JCheckBox was source of event
        if (event.getSource() == checkboxes[count])
            randomCharacters[count].toggle(); // toggle state
    }
}

Execute a Runnable

Shutdown thread pool when threads finish their tasks
public static void main( String args[] )
{
    // create new RandomCharacters object
    RandomCharacters application = new RandomCharacters();

    // set application to end when window is closed
    application.setDefaultCloseOperation( EXIT_ON_CLOSE );
}
} // end class RandomCharacters
23.11 Other Classes and Interfaces in java.util.concurrent

- Callable interface
  - Declares method `call`
  - Method `call` allows a concurrent task to return a value or throw an exception
  - ExecutorService method `submit` takes a Callable and returns a Future representing the result of the task

- Future interface
  - Declares method `get`
  - Method `get` returns the result of the task represented by the Future