UCLA PIC 20A
Java Programming

- **Instructor:** Ivo Dinov,
  Asst. Prof. In Statistics, Neurology and Program in Computing
- **Teaching Assistant:** Yon Seo Kim, PIC

University of California, Los Angeles, Summer 2002
http://www.stat.ucla.edu/~dinov/

Chapter 8 – Threads in Java

- What Is an Thread?
- Example - TimerThread
- Life-cycle of a thread
- Thread Priority
- Synchronizing Threads
- Grouping Threads

What is a Thread?

- A ***thread*** is a single sequential flow of control that runs within a program.
- Ex: A **Web browser** is a multithreaded application – you can scroll a page while it's downloading an applet or an image, play animation and sound concurrently, print a page in the background while you download a new page, or watch three sorting algorithms race to the finish.
- Some books call a thread a ***lightweight process***. A thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

What is a Thread?

- When implementing threads consider using high-level thread API. For example, if your program must perform a task repeatedly, consider using the **java.util.Timer** class. The Timer class is also useful for performing a task after a delay.
- If you're writing a program with a GUI, you might want to use the **javax.swing.Timer** class instead of java.util.Timer. **SwingWorker**, helps you with another common job: performing a task in a background thread, and updating the GUI when the task completes.

What is a Thread?

- Basic support for threads is in the class **java.lang.Thread**. It provides a thread API and provides all the generic behavior for threads.
  - **starting, sleeping, running, yielding, and having a priority.**
- To implement a thread using the Thread class, you need to provide it with a **run** method that performs the thread's task.

TimerTask – example

Using a timer to perform a task after a delay **Reminder.java**

```
import java.util.Timer;//Demo that uses java.util.Timer to
import java.util.TimerTask; //schedule a task to execute once 5 seconds have passed
public class Reminder {
        Timer timer;
        public Reminder(int seconds){
                timer =new Timer();
        timer.schedule(new RemindTask(),seconds*1000);
}
```

TimerTask – example

Using a timer to perform a task after a delay **Reminder.java**

```
class RemindTask extends TimerTask {
    public void run(){ System.out.println("Time's up!");
        timer.cancel();//Terminate the timer thread
    }
}
public static void main (String args []) {
    new Reminder(5);
    System.out.println("Task scheduled.");
}
}
```

1. 2.

TimerTask – example

- Basic components of implementing and scheduling a task a timer thread.
  - Implement a custom subclass of **TimerTask** The run method contains the code that performs the task. Here, the subclass is named **RemindTask** .
  - Create a thread by instantiating the Timer class.
  - Instantiate the timer task object (**RemindTask**()).
  - Schedule the timer task for execution. This example uses the schedule method, with args = timer task; and the delay in milliseconds.

To Stop Timer Threads

- By default, a program keeps running as long as its timer threads are running. To terminate a timer thread:
  - Invoke **cancel** on the timer. You can do this from anywhere in the program, such as from a timer task's run method.
  - Make the timer's thread a "daemon" by creating the timer like this: new Timer(true). If the only threads left in the program are daemon threads, the program exits.
  - After all the timer's scheduled tasks have finished executing, remove all references to the Timer object. Eventually, the timer's thread will terminate.
  - Invoke the System.exit method, which makes the entire program (and all its threads) exit.

To Stop Timer Threads

- Sometimes, timer threads aren't the only threads that can prevent a program from exiting when expected. For example, if you use the AWT at all to make beeps—the AWT automatically creates a nondaemon thread that keeps the program alive. We need to call the System.exit method to make this program.

```
public class ReminderBeep { ……. …..
    public ReminderBeep(int seconds){
        toolkit =Toolkit.getDefaultToolkit();
        timer =new Timer();
        timer.schedule(new
            RemindTask(),seconds*1000);
    }
```


To Stop Timer Threads

We need to call the System.exit method to make this program.

```
class RemindTask extends TimerTask {
    public void run(){
        System.out.println("Time's up!");
        toolkit.beep();
            //timer.cancel();
            //Not necessary since we call System.exit
        System.exit(0);
            //Stops the AWT thread (and everything else)
    }
}
….
} // END:: public class ReminderBeep
```


Performing a task repeatedly

Perform a task once per second.

```
public class AnnoyingBeep {
    Toolkit toolkit;
    Timer timer;
    public AnnoyingBeep(){
        toolkit =Toolkit.getDefaultToolkit();
        timer =new Timer();
        timer.schedule(new RemindTask(), 0,
            1*1000); // initial delay & subsequent rate
    }

class RemindTask extends TimerTask {
    int numWarningBeeps =3;
    public void run(){
        if (numWarningBeeps >0)
            toolkit.beep();
```

```
public static void main(String args [])
{ new Reminder(5);
System.out.println("Task scheduled.");
}
```

Performing a task repeatedly

Perform a task once per second.

```
class RemindTask extends TimerTask {
    int numWarningBeeps =3;
    public void run(){
        if (numWarningBeeps >0) {   toolkit.beep();
            System.out.println("Beep!");
            numWarningBeeps--;
        } else {   toolkit.beep();
        System.out.println("Time's up!");
        //timer.cancel();//Not necessary since we call
        System.exit(0);  //Stops AWT thread/everything
        }
    }
}
...
}
```

Output:

```
Task scheduled.
Beep!
Beep!          //one second after the 1st beep
Beep!          //one second after the 2nd beep
Time's up!     //one second after the 3rd beep
```


Performing a task repeatedly

The AnnoyingBeep program uses a three-argument version of the schedule method to specify that its task should execute once a second, beginning immediately. Here are all the Timer methods you can use to schedule repeated executions of tasks:

- schedule(TimerTask *task* ,long *delay* ,long *period*)
- schedule(TimerTask *task* ,Date *time* ,long *period*)
- scheduleAtFixedRate(TimerTask *task*, long *delay*,long *period*)
- scheduleAtFixedRate(TimerTask *task*,Date *firstTime*,long *period*)

If smoothness is important to schedule a task for repeated execution, use a schedule method; or a **scheduleAtFixedRate** method when time synchronization is important. Ex., AnnoyingBeep program uses the schedule method, which means that the annoying beeps will all be at least 1 second apart. If one beep is late for any reason, all subsequent beeps will be delayed.

Thread's run method

- Customizing the Thread's run method: pp. 277

Class Thread: An Overview of the Thread Methods

- Thread-related methods
  - See API for more details (especially exceptions)
  - Constructors
    - `Thread( threadName )`
    - `Thread()`
      - Creates an auto numbered `Thread` of format `Thread-1`, `Thread-2`...
  - `run`
    - "Does work" of thread
    - Can be overridden in subclass of `Thread` or in `Runnable` object (more on interface `Runnable` in 15.10)
  - `start`
    - Launches thread, then returns to caller
    - Calls `run`
    - Error to call `start` twice for same thread

Class Thread: An Overview of the Thread Methods

- `Thread` methods
  - `static` method `sleep( milliseconds )`
    - Thread sleeps (does not contend for processor) for number of milliseconds
    - Can give lower priority threads a chance to run
  - `interrupt`
    - Interrupts a thread
  - `static` method `interrupted`
    - Returns `true` if current thread interrupted
  - `isInterrupted`
    - Determines if a thread is interrupted
  - `isAlive`
    - Returns `true` if `start` called and thread not dead (`run` has not completed)

Class Thread: An Overview of the Thread Methods

- `Thread` methods
  - `yield` - discussed later
  - `setName( threadName )`
  - `getName`
  - `toString`
    - Returns thread name, priority, and `ThreadGroup` (more 15.11)
  - `static` method `currentThread`
    - Returns reference to currently executing thread
  - `join`
    - Calling thread waits for thread receiving message to die before it can proceed
    - No argument or 0 millisecond argument means thread will wait indefinitely
      - Can lead to deadlock/indefinite postponement

Thread States: Life Cycle of a Thread

- Thread states
  - Born state
    - Thread just created
    - When **`start`** called, enters ready state
  - Ready state (runnable state)
    - Highest-priority ready thread enters running state
  - Running state
    - System assigns processor to thread (thread begins executing)
    - When **`run`** method completes or terminates, enters dead state
  - Dead state
    - Thread marked to be removed by system
    - Entered when **`run`** terminates or throws uncaught exception

Class `Thread`: An Overview of the `Thread` Methods

- Other thread states
  - Blocked state
    - Entered from running state
    - Blocked thread cannot use processor, even if available
    - Common reason for blocked state - waiting on I/O request
  - Sleeping state
    - Entered when **`sleep`** method called
    - Cannot use processor
    - Enters ready state after sleep time expires
  - Waiting state
    - Entered when **`wait`** called in an object thread is accessing
    - One waiting thread becomes ready when object calls **`notify`**
    - **`notifyAll`** - all waiting threads become ready

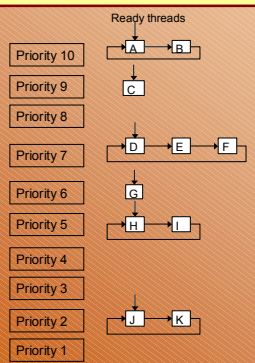
Thread Priorities and Thread Scheduling

- All Java applets / applications are multithreaded
  - Threads have priority from 1 to 10
    - **`Thread.MIN_PRIORITY - 1`**
    - **`Thread.NORM_PRIORITY - 5`** (default)
    - **`Thread.MAX_PRIORITY - 10`**
    - New threads inherit priority of thread that created it
- Timeslicing
  - Each thread gets a quantum of processor time to execute
    - After time is up, processor given to next thread of equal priority (if available)
  - Without timeslicing, each thread of equal priority runs to completion

Thread Priorities and Thread Scheduling

- Java scheduler
  - Keeps highest-priority thread running at all times
  - If timeslicing available, ensure equal priority threads execute in round-robin fashion
  - New high priority threads could postpone execution of lower priority threads
    - Indefinite postponement (starvation)
- Priority methods
  - **`setPriority( int priorityNumber )`**
  - **`getPriority`**
  - **`yield`** - thread yields processor to threads of equal priority
    - Useful for non-timesliced systems, where threads run to completion

Thread Priorities and Thread Scheduling



Thread Priorities and Thread Scheduling

- Example program
  - Demonstrate basic threading techniques
    - Create a class derived from **`Thread`**
    - Use **`sleep`** method
  - Overview
    - Create four threads, which sleep for random amount of time
    - After they finish sleeping, print their name
  - Program has two classes
    - **`PrintThread`**
      - Derives from **`Thread`**
      - Instance variable **`sleepTime`**
    - **`ThreadTester`**
      - Creates four **`PrintThread objects`**

```
1  // ThreadTester.java
2  // Show multiple threads printing at different
3
4  public class ThreadTester {
5     public static void main( String args[] )
6     {
7        PrintThread thread1, thread2, thread3, thread4;
8
9        thread1 = new PrintThread( "thread1" );
10       thread2 = new PrintThread( "thread2" );
11       thread3 = new PrintThread( "thread3" );
12       thread4 = new PrintThread( "thread4" );
13
14       System.err.println( "Th
15
16       thread1.start();
17       thread2.start();
18       thread3.start();
19       thread4.start();
20
21       System.err.println( "Threads started\n" );
22    }
23 }
24
25 class PrintThread extends Thread {
26    private int sleepTime;
27
28    // PrintThread constructor assigns name to thread
29    // by calling Thread constructor
```

● Examples/dinov/ThreadTest.java

main terminates after starting the **PrintThread**s, but the application does not end until the last thread dies.


```
30    public PrintThread( String name )
31    {
32       super( name );
33
34       // sleep between 0 and 5 seconds
35       sleepTime = (int) ( Math.random() * 5000 );
36
37       System.err.println( getName() +
38          " going to sleep with sleepTime: " + sleepTime );
39    }
40
41    // execute the thread
42    public void run()
43    {
44       // put thread to sleep for a random interval
45       try {
46          System.err.println( getName() + " going to
47          Thread.sleep( sleepTime );
48       }
49       catch ( InterruptedException exception ) {
50          System.err.println( exception.toString() );
51       }
52
53       // print thread name
54       System.err.println( getName() + " done sleeping"
55    }
56 }
```

Call superclass constructor to assign name to thread.

start calls the **run** method.

sleep can throw an exception, so it is enclosed in a **try** block.

Starting threads
Threads started

```
thread1 going to sleep
thread2 going to sleep
thread3 going to sleep
thread4 going to sleep
```

```
Name: thread1;  sleep: 3876
Name: thread2;  sleep: 64
Name: thread3;  sleep: 1752
Name: thread4;  sleep: 3120

Starting threads
Threads started

thread2 going to sleep
thread4 going to sleep
thread1 going to sleep
thread3 going to sleep
thread2 done sleeping
thread3 done sleeping
thread4 done sleeping
thread1 done sleeping
```


Synchronizing Threads

- **Monitors**
  - Object with **synchronized** methods
    - Any object can be a monitor
  - Methods declared **synchronized**
    - **public synchronized int myMethod(int x)**
    - Only one thread can execute a **synchronized method** at a time
      - *Obtaining the lock* and *locking* an object
    - If multiple **synchronized** methods, only one may be active
  - Java also has **synchronized** blocks of code

Synchronizing Threads

- Thread may decide it cannot proceed
  - May voluntarily call **wait** while accessing a **synchronized** method
    - Removes thread from contention for monitor object and processor
    - Thread in waiting state
  - Other threads try to enter monitor object
    - Suppose condition first thread needs has now been met
    - Can call **notify** to tell a single waiting thread to enter ready state
    - **notifyAll** - tells all waiting threads to enter ready state

Producer/Consumer Relationship without Thread Synchronization

- Producer / Consumer relationship
  - **Producing** thread may write to buffer (shared memory)
  - **Consuming** thread reads from buffer
  - If not synchronized, data can become corrupted
    - Producer may write before consumer read last data
      - Data lost
    - Consumer may read before producer writes new data
      - Data "doubled"
  - **Using synchronization**
    - If producer knows that consumer has not read last data, calls **wait** (awaits a **notify** command from consumer)
    - If consumer knows producer has not updated data, calls **wait** (awaits **notify** command from producer)

Producer/Consumer Relationship without Thread Synchronization

- Example
  - Producer / Consumer relationship without synchronization
  - Overview
    - Producer writes numbers 1 through 10 to a buffer
    - Consumer reads them from buffer and sums them
    - If producer/consumer operate in order, total should be 55
  - Classes
    - **`ProduceInteger`** and **`ConsumeInteger`**
      - Inherit from **`Thread`**
      - **`sleep`** for random amount of time, then read from / write to buffer
    - **`HoldIntegerUnsynchronized`**
      - Has data and unsynchronized set and get methods
    - **`SharedCell`**
      - Driver, creates threads and calls **`start`**


```
1  // SharedCell.java
2  // Show multiple threads modifying shared object.
3  public class SharedCell {
4     public static void main( String args[] )
5     {
6        HoldIntegerUnsynchronized h =
7           new HoldIntegerUnsynchronized();
8        ProduceInteger p = new ProduceInteger( h );
9        ConsumeInteger c = new ConsumeInteger( h );
10
11       p.start();
12       c.start();
13    }
14 }
15
```

- Class `SharedCell`
- 1. `main`
- 1.1 Initialize objects


```
16 // ProduceInteger.java
17 // Definition of threaded class ProduceInteger
18 public class ProduceInteger extends Thread {
19    private HoldIntegerUnsynchronized pHold;
20
21    public ProduceInteger( HoldIntegerUnsynchronized h )
22    {
23       super( "ProduceInteger" );
24       pHold = h;
25    }
26
27    public void run()
28    {
29       for ( int count = 1; count <= 10; count++ ) {
30          // sleep for a random interval
31          try {
32             Thread.sleep( (int) ( Math.random() * 3000 ) );
33          }
34          catch( InterruptedException e ) {
35             System.err.println( e.toString() );
36          }
37
38          pHold.setSharedInt( count );
39       }
40
41       System.err.println( getName() +
42          " finished producing values" +
43          "\nTerminating " + getName() );
44    }
45 }
46
```

`pHold` refers to a **`HoldIntegerUnsynchronized`** object, and will use its set methods.


```
47 // ConsumeInteger.java
48 // Definition of threaded class ConsumeInteger
49 public class ConsumeInteger extends Thread {
50    private HoldIntegerUnsynchronized cHold;
51
52    public ConsumeInteger( HoldIntegerUnsynchronized h )
53    {
54       super( "ConsumeInteger" );
55       cHold = h;
56    }
57
58    public void run()
59    {
60       int val, sum = 0;
61
62       do {
63          // sleep for a random interval
64          try {
65             Thread.sleep( (int) ( Math.random() * 3000 ) );
66          }
67          catch( InterruptedException e ) {
68             System.err.println( e.toString() );
69          }
70
71          val = cHold.getSharedInt();
72          sum += val;
73       } while ( val != 10 );
74
75       System.err.println(
76          getName() + " retrieved values totaling: " + sum +
77          "\nTerminating " + getName() );
78    }
79 }
```



```
1  // HoldIntegerUnsynchronized.java
2  // Definition of class HoldIntegerUnsynchronized
3  public class HoldIntegerUnsynchronized {
4     private int sharedInt = -1;
5
6     public void setSharedInt( int val )
7     {
8        System.err.println( Thread.currentThread().getName() +
9           " setting sharedInt to " + val );
10       sharedInt = val;
11    }
12
13    public int getSharedInt()
14    {
15       System.err.println( Thread.currentThread().getName() +
16          " retrieving sharedInt value " + sharedInt );
17       return sharedInt;
18    }
19 }
```

Because the set and get methods are unsynchronized, the two threads could call them at the same time.

Class `HoldInteger Unsynchronized`

1. Instance variable
2. `setSharedInt` (unsynchronized)
3. `getSharedInt` (unsynchronized)


```
ConsumeInteger retrieving sharedInt value -1
ConsumeInteger retrieving sharedInt value -1
ProduceInteger setting sharedInt to 1
ProduceInteger setting sharedInt to 2
ConsumeInteger retrieving sharedInt value 2
ProduceInteger setting sharedInt to 3
ProduceInteger setting sharedInt to 4
ProduceInteger setting sharedInt to 5
ConsumeInteger retrieving sharedInt value 5
ProduceInteger setting sharedInt to 6
ProduceInteger setting sharedInt to 7
ProduceInteger setting sharedInt to 8
ConsumeInteger retrieving sharedInt value 8
ConsumeInteger retrieving sharedInt value 8
ProduceInteger setting sharedInt to 9
ConsumeInteger retrieving sharedInt value 9
ConsumeInteger retrieving sharedInt value 9
ProduceInteger setting sharedInt to 10
ProduceInteger finished producing values
Terminating ProduceInteger
ConsumeInteger retrieving sharedInt value 10
ConsumeInteger ret
Terminating Consum
```

- Program Output

Notice how the producer and consumer act out of order, which results in a sum of 49 (not 55).

Producer/Consumer Relationship with Thread Synchronization

- Condition variable of a monitor
  - Variable used to test some condition
    - Determines if thread should call **`wait`**
  - For our producer / consumer relationship
    - Condition variable determines whether the producer should write to buffer or if consumer should read from buffer
    - Use boolean variable **`writeable`**
    - If **`writeable true`**, producer can write to buffer
      - If **`false`**, then producer calls **`wait`**, and awaits **`notify`**
    - If **`writeable false`**, consumer can read from buffer
      - If **`true`**, consumer calls **`wait`**

Producer/Consumer Relationship with Thread Synchronization

- Redo example program with synchronization
  - Synchronize the set and get methods
    - Once the producer writes to memory, **`writeable`** is **`false`** (cannot write again)
    - Once consumer reads, **`writeable`** is **`true`** (cannot read again)
    - Each thread relies on the other to toggle **`writeable`** and call **`notify`**
  - Only class **`HoldIntegerUnsynchronized`** is changed
    - Now called **`HoldIntegerSynchronized`**
    - We only changed the implementation of the set and get methods


```
1  // HoldIntegerSynchronized.java
2  // Definition of class HoldIntegerSynchronized that
3  // uses thread synchronization to ensure that both
4  // threads access sharedInt at the proper times.
5  public class HoldIntegerSynchronized {
6     private int sharedInt = -1;
7     private boolean
8
9     public synchron
10    {
11       while ( !writeable ) {  // not the producer's turn
12          try {
13             wait();
14          }
15          catch ( InterruptedException e ) {
16             e.printStackTrace();
17          }
18       }
19
20       System.err.println( Thread.currentThread().getName() +
21          " setting sharedInt to " + val );
22       sharedInt = val;
23
24       writeable = false;
25       notify();  // tell a waiting thread to
26    }
27
```

Test the condition variable. If it is not the producer's turn, then **`wait`**.

If **`writeable`** is **`true`**, write to the buffer, toggle **`writeable`**, and **`notify`** any waiting threads (so they may read from the buffer).


```
28    public synchronized int getSharedInt()
29    {
30       while ( writeable ) {   // not the consumer's turn
31          try {
32             wait();
33          }
34          catch ( InterruptedException e ) {
35             e.printStackTrace();
36          }
37       }
38
39       writeable = true;
40       notify();  // tell a waiting thread to become ready
41
42       System.err.println( Thread.currentThread().getName() +
43          " retrieving sharedInt value " + sharedInt );
44       return sharedInt;
45    }
46 }
```

As with **`setSharedInt`**, test the condition variable. If not the consumer's turn, then **`wait`**.

If it is ok to read (**`writeable`** is **`false`**), set **`writeable`** to **`true`**, **`notify`**, and **`return sharedInt`**.


```
ProduceInteger setting sharedInt to 1
ConsumeInteger retrieving sharedInt value 1
ProduceInteger setting sharedInt to 2
ConsumeInteger retrieving sharedInt value 2
ProduceInteger setting sharedInt to 3
ConsumeInteger retrieving sharedInt value 3
ProduceInteger setting sharedInt to 4
ConsumeInteger retrieving sharedInt value 4
ProduceInteger setting sharedInt to 5
ConsumeInteger retrieving sharedInt value 5
ProduceInteger setting sharedInt to 6
ConsumeInteger retrieving sharedInt value 6
ProduceInteger setting sharedInt to 7
ConsumeInteger retrieving sharedInt value 7
ProduceInteger setting sharedInt to 8
ConsumeInteger retrieving sharedInt value 8
ProduceInteger setting sharedInt to 9
ConsumeInteger retrieving sharedInt value 9
ProduceInteger setting sharedInt to 10
ProduceInteger finished producing values
Terminating ProduceInteger
ConsumeInteger retrieving sharedInt value 10
ConsumeInteger retrieve
Terminating ConsumeInte
```

- Program Output

The producer and consumer act in order, and the proper total is reached (55).

Producer/Consumer Relationship: The Circular Buffer

- Previous program
  - Does access data properly, but not optimally
  - Producer cannot produce faster than consumer can consume
    - To allow this, use a circular buffer
    - Has enough cells to handle "extra" production
    - Once producer knows consumer has read data, allowed to overwrite it
- Redo program with a circular buffer
  - For the circular buffer, use 5-element array
    - Variables **`readLoc`** and **`writeLoc`** keep track of position in array
    - Incremented, and kept between **`0`** and **`4`** with **`% 5`**
    - Condition variables **`readable`** and **`writeable`**

Producer/Consumer Relationship: The Circular Buffer

- Redo program with a circular buffer
  - Producer starts first, so **`writeLoc > readLoc`** (in beginning)
    - If **`writeLoc == readLoc`** (in set method), producer looped around and "caught up" to consumer
    - Buffer is full, so producer stops writing (**`wait`**)
  - In get method
    - If **`readLoc == writeLoc`** then consumer "caught up" to producer
    - Buffer is empty, so consumer stops reading (**`wait`**)
  - This time, use a GUI
  - Only the set and get methods (in **`HoldIntegerSynchronized`**) change significantly


```
1  // SharedCell.java
2  // Show multiple threads modifying shared object.
3  import java.text.DecimalFormat;
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class SharedCell extends JFrame {
9     public SharedCell()
10    {
11       super( "Demonstrating Thread Synchronization" );
12       JTextArea output = new JTextArea( 20, 30 );
13
14       getContentPane().add( new JScrollPane( output ) );
15       setSize( 500, 500 );
16       show();
17
18       // set up threads and start threads
19       HoldIntegerSynchronized h =
20          new HoldIntegerSynchronized( output );
21       ProduceInteger p = new ProduceInteger( h, output );
22       ConsumeInteger c = new ConsumeInteger( h, output );
23
24       p.start();
25       c.start();
26    }
27
```



```
28    public static void main( String args[] )
29    {
30       SharedCell app = new SharedCell();
31       app.addWindowListener(
32          new WindowAdapter() {
33             public void windowClosing( WindowEvent e )
34             {
35                System.exit( 0 );
36             }
37          }
38       );
39    }
40 }
41
```

- 1. GUI added


```
42 // ProduceInteger.java
43 // Definition of threaded class ProduceInteger
44 import javax.swing.JTextArea;
45
46 public class ProduceInteger extends Thread {
47    private HoldIntegerSynchronized pHold;
48    private JTextArea output;
49
50
51    public ProduceInteger( HoldIntegerSynchronized h,
52                           JTextArea o )
53    {
54       super( "ProduceInteger" );
55       pHold = h;
56       output = o;
57    }
58
59    public void run()
60    {
61       for ( int count = 1; count <= 10; count++ ) {
62          // sleep for a random interval
63          // Note: Interval shortened purposely to fill buffer
64          try {
65             Thread.sleep( (int) ( Math.random() * 500 ) );
66          }
67          catch( InterruptedException e ) {
68             System.err.println( e.toString() );
69          }
70
71          pHold.setSharedInt( count );
72       }
```



```
73
74       output.append( "\n" + getName() +
75          " finished producing values" +
76          "\nTerminating " + getName() + "\n" );
77    }
78 }
79
```

- 1.1 Update GUI


```
80 // ConsumeInteger.java
81 // Definition of threaded class ConsumeInteger
82 import javax.swing.JTextArea;
83
84 public class ConsumeInteger extends Thread {
85    private HoldIntegerSynchronized cHold;
86    private JTextArea output;
87
88    public ConsumeInteger( HoldIntegerSynchronized h,
89                           JTextArea o )
90    {
91       super( "ConsumeInteger" );
92       cHold = h;
93       output = o;
94    }
95
96    public void run()
97    {
98       int val, sum = 0;
99
100      do {
101         // sleep for a random interval
102         try {
103            Thread.sleep( (int) ( Math.random() * 3000 ) );
104         }
105         catch( InterruptedException e ) {
106            System.err.println( e.toString() );
107         }
108
109         val = cHold.getSharedInt();
110         sum += val;
111      } while ( val != 10 );
```

nodate

```
112
113     output.append( "\n" + getName() +
114        " retrieved values totaling: " + sum +
115        "\nTerminating " + getName() + "\n" );
116  }
117}
118
```

- 1.1 Update GUI


```
119// HoldIntegerSynchronized.java
120// Definition of class HoldIntegerSynchronized that
121// uses thread synchronization to ensure that both
122// threads access sharedInt at the proper times.
123import javax.swing.JTextArea;
124import java.text.DecimalFormat;
125
126public class HoldIntegerSynchronized {
127   private int sharedInt[] = { -1, -1, -1, -1, -1 };
128   private boolean writeable = true;
129   private boolean readable = false;
130   private int readLoc = 0, writeLoc = 0;
131   private JTextArea output;
132
133   public HoldIntegerSynchronized( JTextArea
134   {
135      output = o;
136   }
137
138   public synchronized void setSharedInt( int val )
139   {
140      while ( !writeable ) {
141         try {
142            output.append( " WAITING TO PRODUCE " + val );
143            wait();
144         }
145         catch ( InterruptedException e ) {
146            System.err.println( e.toString() );
147         }
148      }
149
```

Notice all the added instance variables, including the circular buffer and condition variables.


```
150      sharedInt[ writeLoc ] = val;
151      readable = true;
152
153      output.append( "\nProduced " + val +
154                     " into cell " + writeLoc );
155
156      writeLoc = ( writeLoc + 1 ) % 5;
157
158      output.append( "\twrite " + writeLoc +
159                     "\tread " + readLoc);
160      displayBuffer( output, sharedInt );
161
162      if ( writeLoc == readLoc ) {
163         writeable = false;
164         output.append( "\nBUFFER FULL" );
165      }
166
167      notify();
168   }
169
170   public synchronized int getSharedInt()
171   {
172      int val;
173
174      while ( !readable ) {
175         try {
176            output.append( " WAITING TO CONSUME" );
177            wait();
178         }
179         catch ( InterruptedException e ) {
180            System.err.println( e.toString() );
181         }
182      }
```

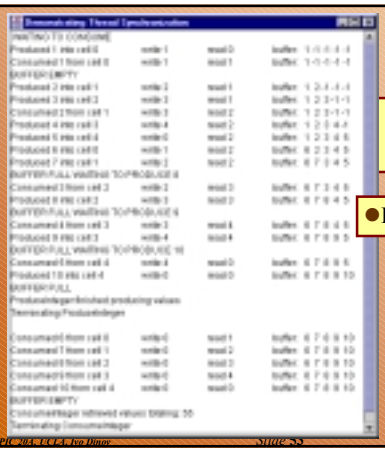
Set appropriate location in the circular buffer. Update **`readable`**.

Increment **`writeLoc`**, use **`% 5`** to keep it in range.

Test for full buffer.


```
183
184      writeable = true;
185      val = sharedInt[ readLoc ];
186
187      output.append( "\nConsumed " + val +
188                     " from cell " + readLoc );
189
190      readLoc = ( readLoc + 1 ) % 5;
191
192      output.append( "\twrite " + writeLoc +
193                     "\tread " + readLoc );
194      displayBuffer( output, sharedInt );
195
196      if ( readLoc == writeLoc ) {
197         readable = false;
198         output.append( "\nBUFFER EMPTY" );
199      }
200
201      notify();
202
203      return val;
204   }
205
206   public void displayBuffer( JTextArea out, int buf[] )
207   {
208      DecimalFormat formatNumber = new DecimalFormat( " #;-#" );
209      output.append( "\tbuffer: " );
210
211      for ( int i = 0; i < buf.length; i++ )
212         out.append( " " + formatNumber.format( buf[ i ] ));
213   }
214}
```

Similar to **`setSharedInt`**. Update **`readLoc`**, test for empty buffer, **`return val`**.



- Program Output

Daemon Threads

- Daemon threads
  - Threads that run for benefit of other threads
    - Garbage collector
  - Run in background
    - Use processor time that would otherwise go to waste
  - Unlike normal threads, do not prevent a program from terminating
    - When only daemon threads remain, program exits
  - Must designate a thread as daemon before **`start`** called
    `setDaemon( true );`
  - Method **`isDaemon`**
    - Returns **`true`** if thread is a daemon thread

Runnable Interface

- Java does not support multiple inheritance
  - Instead, use interfaces
  - Until now, inherited from class **Thread**, overrode **run**
- Multithreading for an already derived class
  - Implement interface **Runnable** (**java.lang**)
    - New class objects "are" **Runnable** objects
  - Override **run** method
    - Controls thread, just as deriving from **Thread** class
    - In fact, class **Thread** implements interface **Runnable**
  - Create new threads using **Thread** constructors
    - **Thread(runnableObject)**
    - **Thread(runnableObject, threadName)**

Runnable Interface

- Synchronized blocks of code

```
synchronized( monitorObject ){
   ...
}
```

  - **monitorObject**- Object to be locked while thread executes block of code
- Suspending threads
  - In earlier versions of Java, there were methods to suspend/resume threads
    - Dangerous, can lead to deadlock
  - Instead, use **wait** and **notify**

Runnable Interface

- Upcoming example program
  - Create a GUI and three threads, each constantly displaying a random letter
  - Have suspend buttons, which will suspend a thread
    - Actually calls **wait**
    - When suspend unclicked, calls **notify**
    - Use an array of **boolean**s to keep track of which threads are suspended


```
1  // RandomCharacters.java
2  // Demonstrating the Runnableinterface
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class RandomCharacters extends JApplet
8                              implements Runnable,
9                                         ActionListener {
10   private String alphabet = "ABCDEFGHIJ
11   private JLabel outputs[];
12   private JCheckBox checkboxes[];
13   private final static int SIZE = 3;
14
15   private Thread threads[];
16   private boolean suspended[];
17
18   public void init()
19   {
20      outputs = new JLabel[ SIZE ];
21      checkboxes = new JCheckBox[ SIZE ];
22
23      threads = new Thread[ SIZE ];
24      suspended = new boolean[ SIZE ];
25
26      Container c = getContentPane();
27      c.setLayout( new GridLayout( SIZE, 2, 5, 5 ) );
28
```

Use a **boolean** array to keep track of which threads are "suspended". We will actually use **wait** and **notify** to suspend the threads.


```
29      for ( int i = 0; i < SIZE; i++ ) {
30         outputs[ i ] = new JLabel();
31         outputs[ i ].setBackground( Color.green );
32         outputs[ i ].setOpaque( true );
33         c.add( outputs[ i ] );
34
35         checkboxes[ i ] = new JCheckBox( "Sus
36         checkboxes[ i ].addActionListener( th
37         c.add( checkboxes[ i ] );
38      }
39   }
40
41   public void start()
42   {
43      // create threads and start every time start is called
44      for ( int i = 0; i < threads.length; i++ ) {
45         threads[ i ] =
46            new Thread( this, "Thread " +
47         threads[ i ].start();
48      }
49   }
50
51   public void run()
52   {
53      Thread currentThread = Thread.currentThread();
54      int index = getIndex( currentThread );
55      char displayChar;
56
57      while ( threads[ index ] == currentThread ) {
58         // sleep from 0 to 1 second
59         try {
60            Thread.sleep( (int) ( Math.random() * 1000 ) );
```

Use the **Thread** constructor to create new threads. The **Runnable** object is **this** applet.

Loop will execute indefinitely because **threads[index] == currentThread**. The **stop** method in the applet sets all threads to **null**.

start calls **run** method for thread.


```
62            synchronized( this ) {
63               while ( suspended[ index ] &&
64                       threads[ index ] == currentThread )
65                  wait();
66            }
67         }
68         catch ( InterruptedException e ) {
69            System.err.println( "sleep i
70         }
71
72         displayChar = alphabet.charAt(
73                          (int) ( Math.random() * 26 ) );
74
75         outputs[ index ].setText( currentThread.getName() +
76            ": " + displayChar );
77      }
78
79      System.err.println(
80         currentThread.getName() + " terminating" );
81   }
82
83   private int getIndex( Thread current )
84   {
85      for ( int i = 0; i < threads.length; i++ )
86         if ( current == threads[ i ] )
87            return i;
88
89      return -1;
90   }
91
```

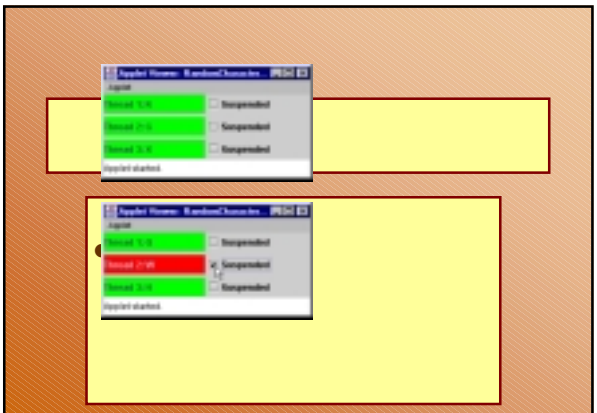
Synchronized block tests suspended array to see if a thread should be "suspended". If so, calls **wait**.

```
92    public synchronized void stop()
93    {
94       // stop threads every time stop is called
95       // as the user browses another Web page
96       for ( int i = 0; i < threads.length; i++ )
97          threads[ i ] = null;
98
99       notifyAll();
100   }
101
102   public synchronized void actionPerformed( ActionEvent e )
103   {
104      for ( int i = 0; i < checkboxes.length; i++ ) {
105         if ( e.getSource() == checkboxes[ i ] ) {
106            suspended[ i ] = !suspended[ i ];
107
108            outputs[ i ].setBackground(
109               !suspended[ i ] ? Color.green : Color.red );
110
111            if ( !suspended[ i ] )
112               notify();
113
114            return;
115         }
116      }
117   }
118}
```

Sets all threads to **`null`**, which causes loop in **`run`** to end, and **`run`** terminates.

Loop and find which box was checked, and suspend appropriate thread. The **`run`** method checks for suspended threads.

If suspend is off, then notify the appropriate thread.



Thread Groups

- Thread groups
  - Threads in a thread group can be dealt with as a group
    - ❑May want to **`interrupt`** all threads in a group
  - Thread group can be parent to a child thread group
- Class **`ThreadGroup`**
  - Constructors

    `ThreadGroup( threadGroupName )`

    `ThreadGroup( parentThreadGroup, name )`
    - ❑Creates child **`ThreadGroup`** named **`name`**

Thread Groups

- Associating **`Thread`**s with **`ThreadGroup`**s
  - Use constructors
  - **`Thread( threadGroup, threadName )`**
  - **`Thread( threadGroup, runnableObject )`**
    - ❑Invokes **`run`** method of **`runnableObject`** when thread executes
  - **`Thread( threadGroup, runnableObject, threadName )`**
    - ❑As above, but **`Thread`** named **`threadName`**

Thread Groups

- **`ThreadGroup`** Methods
  - See API for more details
  - **`activeCount`**
    - ❑Number of active threads in a group and all child groups
  - **`enumerate`**
    - ❑Two versions copy active threads into an array of references
    - ❑Two versions copy active threads in a child group into an array of references
  - **`getMaxPriority`**
    - ❑Returns maximum priority of a **`ThreadGroup`**
    - ❑**`setMaxPriority`**
  - **`getName, getParent`**