Chapter 8 – Threads in Java

What Is a Thread?
- A thread is a single sequential flow of control that runs within a program.
- Example - TimerThread
- Life-cycle of a thread
- Thread Priority
- Synchronizing Threads
- Grouping Threads

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`
```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 ReminderTask(), seconds*1000);
    }
}
```
TimerTask – example

Using a timer to perform a task after a delay

```java
class ReminderTask 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.");
    }
}
```

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.

Performing a task repeatedly

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

```java
class ReminderTask 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.
```java
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
```

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
    - Schedules thread, then returns to caller
    - Calls run
    - Error to call start twice for same thread

- Thread methods
  - `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
  - `isInterrupted`
    - Returns true if current thread interrupted
  - `isAlive`
    - Returns true if start called and thread not dead (run has not completed)

- yield - discussed later
- `getName()`
- `getName()`
- `tostring()`

- `currentThread`
- `toString()`
- `getCurrentThread()`
- `toString()`
Thread States: Life Cycle of a Thread

- **Thread states**
  - Born state
    - 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

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

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 goes to next thread of equal priority (if available)
  - Without timeslicing, each thread of equal priority runs to completion

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
Thread Tester

```java
public class ThreadTester {
    public static void main(String[] args) {
        PrintThread thread1, thread2, thread3, thread4;

        thread1 = new PrintThread("thread1");
        thread2 = new PrintThread("thread2");
        thread3 = new PrintThread("thread3");
        thread4 = new PrintThread("thread4");

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

        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();

        System.err.println("Threads started");
    }
}
```

### PrintThread Class

```java
class PrintThread extends Thread {
    private int sleepTime;

    // PrintThread constructor assigns name to thread
    public PrintThread(String name) {
        super(name);

        // sleep between 0 and 5 seconds
        sleepTime = (int) (Math.random() * 5000);

        System.err.println("Name: "+getName()+"; sleep: "+sleepTime);
    }

    // execute the thread
    public void run() {
        try {
            System.err.println(getName()+" going to sleep");
            Thread.sleep(sleepTime);
        } catch (InterruptedException exception) {
            System.err.println(exception.toString());
        }

        System.err.println(getName()+" 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 `synchronized` method at a time
  - Java also has `synchronized` blocks of code

- **Producer/Consumer Relationship without Thread Synchronization**
  - Producer may write to buffer (shared memory)
  - Consumer reads from buffer
  - If not synchronized, data can become corrupted
    - Producer may write before consumer reads last data
    - Consumer may read before producer writes new data
    - Data "corrupted"
  - Using synchronization
    - Producer & consumer need to wait and notify

```java
// ThreadTester.java
// Show multiple threads sleeping at different intervals.
//
// Example of thread synchronisation
//
public class ThreadTester {
    public static void main(String[] args) {
        PrintThread thread1, thread2, thread3, thread4;

        thread1 = new PrintThread("thread1");
        thread2 = new PrintThread("thread2");
        thread3 = new PrintThread("thread3");
        thread4 = new PrintThread("thread4");

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

        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();

        System.err.println("Threads started");
    }
}
```

```java
// PrintThread.java
// Synchronizing threads
//
// Producer/Consumer relationship
//
public PrintThread(String name) {
    super(name);

    // sleep between 0 and 5 seconds
    sleepTime = Math.random() * 5000;

    System.err.println("Start call for run method");

    // sleep can throw an exception, so it
    // is enclosed in a try block.
    try {
        System.err.println(getName()+" going to sleep");
        Thread.sleep(sleepTime);
    } catch (InterruptedException exception) {
        System.err.println(exception.toString());
    }

    System.err.println(getName()+" done sleeping");
}
```
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`
    - Extends `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`

---

**Program Output**

```
Terminating ProduceInteger
ProduceInteger setting sharedInt to 10
ProduceInteger setting sharedInt to 9
ProduceInteger setting sharedInt to 8
ProduceInteger setting sharedInt to 7
ProduceInteger setting sharedInt to 6
ProduceInteger setting sharedInt to 5
ProduceInteger setting sharedInt to 4
ProduceInteger setting sharedInt to 3
ProduceInteger setting sharedInt to 2
ProduceInteger setting sharedInt to 1
ConsumeInteger retrieving sharedInt value = 1
ConsumeInteger retrieving sharedInt value = 2
ConsumeInteger retrieving sharedInt value = 3
ConsumeInteger retrieving sharedInt value = 4
ConsumeInteger retrieving sharedInt value = 5
ConsumeInteger retrieving sharedInt value = 6
ConsumeInteger retrieving sharedInt value = 7
ConsumeInteger retrieving sharedInt value = 8
ConsumeInteger retrieving sharedInt value = 9
ConsumeInteger retrieving sharedInt value = 10
```

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 consumer should read from buffer
    - Use boolean variable `writeable`
    - If `writeable true`, producer can write to buffer
      - Otherwise, then producer calls `wait`, and awaits `notify`
    - If `writeable false`, consumer can read from buffer
      - Otherwise, consumer calls `wait`

Producer/Consumer Relationship

- 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

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

Program Output

- Producer setting `sharedInt to 1`
- Consumer retrieving `sharedInt value 1`
- Producer setting `sharedInt to 2`
- Consumer retrieving `sharedInt value 2`
- Producer setting `sharedInt to 3`
- Consumer retrieving `sharedInt value 3`
- Producer setting `sharedInt to 4`
- Consumer retrieving `sharedInt value 4`
- Producer setting `sharedInt to 5`
- Consumer retrieving `sharedInt value 5`
- Producer setting `sharedInt to 6`
- Consumer retrieving `sharedInt value 6`
- Producer setting `sharedInt to 7`
- Consumer retrieving `sharedInt value 7`
- Producer setting `sharedInt to 8`
- Consumer retrieving `sharedInt value 8`
- Producer setting `sharedInt to 9`
- Consumer retrieving `sharedInt value 9`
- Producer setting `sharedInt to 10`
- Consumer retrieving `sharedInt value 10`
- Producer finished producing values
- Terminating Producer
- Consumer retrieving `sharedInt value 10`
- Consumer retrieves this producer and consumer act in order, and the proper total is reached (55).
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
  - This time, use a GUI
  - Only the set and get methods (in HoldIntegerSynchronized) change significantly

```
1 // SharedCell.java
2 // New multiple threads modifying shared object.
3 import java.text.DecimalFormat;
4 import java.awt.
5 import java.awt.event.
6 import javax.swing.

7 public class SharedCell extends JFrame {
8     // Show multiple threads modifying shared object.
9     public SharedCell() {
10         super( "Demonstrating Thread Synchronization" );
11         JTextArea output = new JTextArea( 20, 30 );
12         getContentPane().add( new JScrollPane( output ) );
13         setSize( 500, 500 );
14         show();
15
16         // set up threads and start threads
17         HoldIntegerSynchronized h = new HoldIntegerSynchronized( output );
18         ProduceInteger p = new ProduceInteger( h, output );
19         ConsumeInteger c = new ConsumeInteger( h, output );
20         p.start();
21         c.start();
22     }
23
24     public static void main( String args[] ) {
25         SharedCell app = new SharedCell();
26         app.addWindowListener( new WindowAdapter() {
27             public void windowClosing( WindowEvent e ) {
28                 System.exit( 0 );
29             }
30         } );
31     }
32 }
```

```
1 // ProduceInteger.java
2 // Definition of threaded class ProduceInteger
3 import javax.swing.JTextArea;

4 public class ProduceInteger extends Thread {
5     private HoldIntegerSynchronized pHold;
6     private JTextArea output;

7     public ProduceInteger( HoldIntegerSynchronized h, JTextArea o ) {
8         super( "ProduceInteger" );
9         pHold = h;
10         output = o;
11     }
12
13     public void run() {
14         for ( int count = 1; count <= 10; count++ ) {
15             try {
16                 Thread.sleep( (int) ( Math.random() * 500 ) );
17             } catch( InterruptedException e ) {
18                 System.err.println( e.toString() );
19             }
20             pHold.setSharedInt( count );
21         }
22         output.append( " ' + getName() + ", sum = 0;
23         output.append( "Finished producing values = " + sum );
24         output.append( "' + getName() + "\n" );
25     }
26 }
```

```
1 // ConsumeInteger.java
2 // Definition of threaded class ConsumeInteger
3 import javax.swing.JTextArea;

4 public class ConsumeInteger extends Thread {
5     private HoldIntegerSynchronized cHold;
6     private JTextArea output;

7     public ConsumeInteger( HoldIntegerSynchronized h, JTextArea o ) {
8         super( "ConsumeInteger" );
9         cHold = h;
10         output = o;
11     }
12
13     public void run() {
14         int val, sum = 0;
15         do { // sleep for a random interval
16             try { Thread.sleep( (int) ( Math.random() * 3000 ) );
17                 catch( InterruptedException e ) { System.err.println( e.toString() );
18             }
19             val = cHold.getSharedInt();
20             sum += val;
21         } while( val != 10 );
22     }
23 }
```

1.1 Update GUI

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

Class HoldInteger

1. Instance variables

2. `setSharedInt`

3. `getSharedInt`

4. GUI method

```java
126 import javax.swing.JTextArea;
127 import java.text.DecimalFormat;
128
129 public class HoldIntegerSynchronized {
130
131 private int sharedInt[] = { -1, -1, -1, -1, -1 };
132 private boolean writeable = true;
133 private boolean readable = false;
134 private int readLoc = 0, writeLoc = 0;
135 private JTextArea output;
136
137 public HoldIntegerSynchronized( JTextArea o )
138 {
139 output = o;
140 }
141
142 public synchronized void setSharedInt( int val )
143 {
144 while ( !writeable ) {
145 try {
146 output.append( " WAITING TO PRODUCE " + val );
147 wait();
148 }
149 catch ( InterruptedException e ) {
150 System.err.println( e.toString() );
151 }
152 }
153
154 sharedInt[ writeLoc ] = val;
155 readable = true;
156
157 output.append( "Produced " + val + " into cell " + writeLoc );
158
159 writeLoc = ( writeLoc + 1 ) % 5;
160 output.append( "	write " + writeLoc + "	read " + readLoc );
161 displayBuffer( output, sharedInt );
162
163 if ( writeLoc == readLoc ) {
164 writeable = false;
165 output.append( "BUFFER FULL" );
166 }
167
168 notify();
169 }
170
171 public synchronized int getSharedInt()
172 {
173 int val;
174
175 while ( !readable ) {
176 try {
177 output.append( " WAITING TO CONSUME" );
178 wait();
179 }
180 catch ( InterruptedException e ) {
181 System.err.println( e.toString() );
182 }
183 }
184
185 writeable = true;
186 val = sharedInt[ readLoc ];
187
188 output.append( "Consumed " + val + " from cell " + readLoc );
189
190 readable = true;
191 readLoc = ( readLoc + 1 ) % 5;
192
193 output.append( "	write " + writeLoc + "	read " + readLoc );
194 displayBuffer( output, sharedInt );
195
196 if ( readLoc == writeLoc ) {
197 readable = false;
198 output.append( "BUFFER 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( "	buffer: " );
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`.

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

---

### 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

- **Method isDaemon**
  - Return `true` if thread is a daemon thread

- **Program Output**

```java
188 output.append( "WAITING TO CONSUME" );
189 wait();
190 output.append( "WAITING TO PRODUCE " + val );
191 wait();
192 output.append( "PRODUCED " + val + " into cell " + writeLoc );
193
194 writeable = true;
195 val = sharedInt[ writeLoc ];
196
197 output.append( "produced " + val + " into cell " + writeLoc );
198
199 output.append( "write " + writeLoc + "	read " + readLoc );
200 displayBuffer( output, sharedInt );
201
202 if ( writeLoc == readLoc ) {
203 writeable = false;
204 output.append( "BUFFER FULL" );
205 }
206
207 notify();
208
209 return val;
210 }
211
212 public void displayBuffer( JTextArea out, int buf[]) {
213 }
214 }
```

```java
185 writeable = true;
186 val = sharedInt[ readLoc ];
187
188 output.append( "Consumed " + val + " from cell " + readLoc );
189
190 readable = true;
191 readLoc = ( readLoc + 1 ) % 5;
192
193 output.append( "	write " + writeLoc + "	read " + readLoc );
194 displayBuffer( output, sharedInt );
195
196 if ( readLoc == writeLoc ) {
197 readable = false;
198 output.append( "BUFFER EMPTY" );
199 }
200
201 notify();
202
203 return val;
204 }
205
206 public void displayBuffer( JTextArea out, int buf[]) {
207 }
208 ```

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

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

- Java does not support multiple inheritance
  - Instead, use interfaces
  - Until now, inherited from class `Thread`, override `run`
- Multithreading for an already derived class
  - Implement interface `Runnable`
    - New class objects are `Runnable`
  - 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

- 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` to keep track of which threads are suspended

```java
Runnable Interface

```
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 Threads with ThreadGroups
  - 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

- 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
- getName, getParent