Week 4 – Multithreading

Outline

4.1  Introduction
4.2  Class Thread: An Overview of the Thread Methods
4.3  Thread States: Life Cycle of a Thread
4.4  Thread Priorities and Thread Scheduling
4.5  Thread Synchronization
4.6  Producer/Consumer Relationship without Thread Synchronization
4.7  Producer/Consumer Relationship with Thread Synchronization
4.8  Producer/Consumer Relationship: The Circular Buffer
4.9  Daemon Threads
4.10  Runnable Interface
4.11  Thread Groups
4.12  (Optional Case Study) Thinking About Objects: Multithreading
4.13  (Optional) Discovering Design Patterns: Concurrent Design Patterns
4.1 Introduction

• Concurrency normally available in OS primitives
• Java provides built-in multithreading
  – Multithreading improves the performance of some programs
4.2 Class Thread: An Overview of the Thread Methods

- Class **Thread** constructors
  
  ```java
  public Thread( String threadName )
  public Thread()
  ```

- Code for thread in thread’s **run** method

- Method **sleep** makes thread inactive

- Method **interrupt** interrupts a running thread

- Method **isAlive** checks status of a thread

- Method **setName** sets a thread’s name

- Method **join**
  
  - Waits for thread to finish and continues from current thread
4.3 Threaded States: Life Cycle of a Thread

- **Thread states**
  - Born state
    - Thread was just created
  - Ready state
    - Thread’s `start` method invoked
    - Thread can now execute
  - Running state
    - Thread is assigned a processor and running
  - Dead state
    - Thread has completed or exited
    - Eventually disposed of by system
4.3 Thread States: Life Cycle of a Thread

Fig. 4.1 State diagram showing the Life cycle of a thread.
4.4 Thread Priorities and Thread Scheduling

• Java thread priority
  – Priority in range 1-10

• Timeslicing
  – Each thread assigned time on the processor by quantum
  – Keeps highest priority threads running
public class ThreadTester {

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

        // create four PrintThread objects
        thread1 = new PrintThread( "thread1" );
        thread2 = new PrintThread( "thread2" );
        thread3 = new PrintThread( "thread3" );
        thread4 = new PrintThread( "thread4" );

        System.err.println( "\nStarting threads" );

        // start executing PrintThreads
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();

        System.err.println( "Threads started\n" );
    }
}

Class ThreadTester creates four PrintThreads and calls their start methods.
// Each object of this class picks a random sleep interval.
// When a PrintThread executes, it prints its name, sleeps,
// prints its name again and terminates.

class PrintThread extends Thread {
    private int sleepTime;

    // PrintThread constructor assigns name to thread
    // by calling superclass Thread constructor
    public PrintThread( String name )
    {
        super( name );

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

        // display name and sleepTime
        System.err.println(
            "Name: " + getName() + ";  sleep: " + sleepTime );
    }

    // control thread's execution
    public void run()
    {
        // put thread to sleep for a random interval
        try {
            System.out.println( getName() + " going to sleep" );

            // put thread to sleep
            Thread.sleep( sleepTime );
        }
    }
}
60 // if thread interrupted during sleep, catch exception
61 // and display error message
62 catch ( InterruptedException interruptedException ) {
63   System.err.println( interruptedException.toString() );
64 }
65
66 // print thread name
67 System.err.println( getName() + " done sleeping" );
68 }
69
70 } // end class PrintThread
Name: thread1; sleep: 3593
Name: thread2; sleep: 2653
Name: thread3; sleep: 4465
Name: thread4; sleep: 1318

Starting threads
Threads started

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

Name: thread1; sleep: 2753
Name: thread2; sleep: 3199
Name: thread3; sleep: 2797
Name: thread4; sleep: 4639

Starting threads
Threads started

thread1 going to sleep
thread2 going to sleep
thread3 going to sleep
thread4 going to sleep
thread1 done sleeping
thread3 done sleeping
thread2 done sleeping
thread4 done sleeping
4.5 Thread Synchronization

- Java uses monitors for thread synchronization
- The `synchronized` keyword
  - Every `synchronized` method of an object has a monitor
  - One thread inside a `synchronized` method at a time
  - All other threads block until method finishes
  - Next highest priority thread runs when method finishes
4.6 Producer/Consumer Relationship without Synchronization

• Buffer
  – Shared memory region

• Producer thread
  – Calls produce method to add item to buffer
  – Calls *wait* if consumer has not read last message in buffer
  – Writes to empty buffer and calls *notify* for consumer

• Consumer thread
  – Reads message from buffer
  – Calls *wait* if buffer empty

• Synchronize threads to avoid corrupted data
1    // Fig. 4.4: ProduceInteger.java
2    // Definition of threaded class ProduceInteger
3    public class ProduceInteger extends Thread {
4        private HoldIntegerUnsynchronized sharedObject;
5
6        // initialize ProduceInteger thread object
7        public ProduceInteger( HoldIntegerUnsynchronized shared )
8        {
9            super( "ProduceInteger" );
10           sharedObject = shared;
11        }
12
13        // ProduceInteger thread loops 10 times and calls
14        // sharedObject's setSharedInt method each time
15        public void run()
16        {
17           for ( int count = 1; count <= 10; count++ ) {
18                // sleep for a random interval
19                try {
20                    Thread.sleep( ( int ) ( Math.random() * 3000 ) );
21                }
22
23           // process InterruptedException during sleep
24           catch( InterruptedException exception ) {
25                System.err.println( exception.toString() );
26           }
27
28           // call sharedObject method from this
29           // thread of execution
30           sharedObject.setSharedInt( count );
31        }
33 System.err.println(
34     getName() + " finished producing values" +
35     "\nTerminating " + getName() );
36 }
37 }
38 } // end class ProduceInteger
// Fig. 4.5: ConsumeInteger.java
// Definition of threaded class ConsumeInteger
public class ConsumeInteger extends Thread {
    private HoldIntegerUnsynchronized sharedObject;

    // initialize ConsumerInteger thread object
    public ConsumeInteger( HoldIntegerUnsynchronized shared )
    {
        super( "ConsumeInteger" );
        sharedObject = shared;
    }

    // ConsumerInteger thread loops until it receives 10
    // from sharedObject's getSharedInt method
    public void run()
    {
        int value, sum = 0;
        do {
            // sleep for a random interval
            try {
                Thread.sleep( (int) ( Math.random() * 3000 ) );
            }
            // process InterruptedException during sleep
            catch( InterruptedException exception ) {
                System.err.println( exception.toString() );
            }
            value = sharedObject.getSharedInt();
            sum += value;
        } while ( value != 10 );
    }
}
System.err.println(
    getName() + " retrieved values totaling: " + sum + 
    "\nTerminating " + getName() );
}  // end class ConsumeInteger
// Fig. 4.6: HoldIntegerUnsynchronized.java
// Definition of class HoldIntegerUnsynchronized.
public class HoldIntegerUnsynchronized {
    private int sharedInt = -1;

    // unsynchronized method to place value in sharedInt
    public void setSharedInt( int value )
    {
        System.err.println( Thread.currentThread().getName() + 
            " setting sharedInt to " + value );
        sharedInt = value;
    }

    // unsynchronized method return sharedInt's value
    public int getSharedInt()
    {
        System.err.println( Thread.currentThread().getName() + 
            " retrieving sharedInt value " + sharedInt );
        return sharedInt;
    }
}

// end class HoldIntegerUnsynchronized
public class SharedCell {

    // execute application
    public static void main( String args[] )
    {
        HoldIntegerUnsynchronized sharedObject =
            new HoldIntegerUnsynchronized();

        // create threads
        ProduceInteger producer =
            new ProduceInteger( sharedObject );
        ConsumeInteger consumer =
            new ConsumeInteger( sharedObject );

        // start threads
        producer.start();
        consumer.start();
    }
} // end class SharedCell
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 retrieved values totaling: 49
Terminating ConsumeInteger

Output of numbers is not properly synchronized
4.7 Producer/Consumer Relationship with Thread Synchronization

- Synchronize threads to ensure correct data
public class ProduceInteger extends Thread {
    private HoldIntegerSynchronized sharedObject;

    // initialize ProduceInteger thread object
    public ProduceInteger( HoldIntegerSynchronized shared ) {
        super( "ProduceInteger" );
        sharedObject = shared;
    }

    // ProduceInteger thread loops 10 times and calls
    // sharedObject's setSharedInt method each time
    public void run() {
        for ( int count = 1; count <= 10; count++ ) {
            // sleep for a random interval
            try {
                Thread.sleep( ( int ) ( Math.random() * 3000 ) );
            } catch( InterruptedException exception ) { System.err.println( exception.toString() ); }

            // call sharedObject method from this
            // thread of execution
            sharedObject.setSharedInt( count );
        }
    }
}
34       System.err.println(
35             getName() + " finished producing values" +
36             "\nTerminating " + getName() );
37     }
38
39   } // end class ProduceInteger
```java
public class ConsumeInteger extends Thread {
    private HoldIntegerSynchronized sharedObject;
    
    public ConsumeInteger( HoldIntegerSynchronized shared ) {
        super( "ConsumeInteger" );
        sharedObject = shared;
    }

    public void run() {
        int value, sum = 0;
        do {
            // sleep for a random interval
            try {
                Thread.sleep( (int) ( Math.random() * 3000 ) );
            } catch( InterruptedException exception ) {
                System.err.println( exception.toString() );
            }
            // process InterruptedException during sleep
            value = sharedObject.getSharedInt();
            sum += value;
        } while ( value != 10 );
    }
}
```
35 System.err.println(
36     getName() + " retrieved values totaling: " + sum +
37     "\nTerminating " + getName() );
39 }
40 }  // end class ConsumeInteger

Thread prints that it is done consuming
```java
    // Fig. 4.10: HoldIntegerSynchronized.java
    // Definition of class HoldIntegerSynchronized that
    // uses thread synchronization to ensure that both
    // threads access sharedInt at the proper times.
    public class HoldIntegerSynchronized {
        private int sharedInt = -1;
        private boolean writeable = true; // condition variable

        // synchronized method allows only one thread at a time to
        // invoke this method to set the value for a particular
        // HoldIntegerSynchronized object
        public synchronized void setSharedInt( int value )
        {
            while ( !writeable ) { // not the producer's turn
                // thread that called this method must wait
                try {
                    wait();
                } catch ( InterruptedException exception ) {
                    exception.printStackTrace();
                }
            }

            System.err.println( Thread.currentThread().getName() +
                " setting sharedInt to " + value );

            // set new sharedInt value
            sharedInt = value;
        }
    }
```
// indicate that producer cannot store another value until
// a consumer retrieve current sharedInt value
writeable = false;

// tell a waiting thread to become ready
notify();

// synchronized method allows only one thread at a time to
// invoke this method to get the value for a particular
// HoldIntegerSynchronized object
public synchronized int getSharedInt()
{
    while ( writeable ) {  // not the consumer's turn
        // thread that called this method must wait
        try {
            wait();
        }
        // process Interrupted exception while thread waiting
        catch ( InterruptedException exception ) {
            exception.printStackTrace();
        }
    }

    // indicate that producer cant store another value
    // because a consumer just retrieved sharedInt value
    writeable = true;

    // tell a waiting thread to become ready
    notify();

    System.err.println( Thread.currentThread().getName() + 
    " retrieving sharedInt value " + sharedInt );
return sharedInt;
}

}  // end class HoldIntegerSynchronized
sharedcell.java

1 // Fig. 4.11: SharedCell.java
2 // Show multiple threads modifying shared object.
3 public class SharedCell {
4
5    // execute application
6    public static void main( String args[] )
7    {
8        HoldIntegerSynchronized sharedObject =
9            new HoldIntegerSynchronized();
10
11        // create threads
12        ProduceInteger producer =
13            new ProduceInteger( sharedObject );
14        ConsumeInteger consumer =
15            new ConsumeInteger( sharedObject );
16
17        // start threads
18        producer.start();
19        consumer.start();
20    }
21
22 } // end class SharedCell
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 retrieved values totaling: 55
Terminating ConsumeInteger

Output of numbers is properly synchronized
4.8 Producer/Consumer Relationship: The Circular Buffer

- Circular buffer
  - Multiple memory cells
  - Produce item if one or more empty cells
  - Consume item if one or more filled cells
```java
// Fig. 4.12: UpdateThread.java
// Class for updating JTextArea with output.

// Java extension packages
import javax.swing.*;

public class UpdateThread extends Thread {
    private JTextArea outputArea;
    private String messageToOutput;

    // initialize outputArea and message
    public UpdateThread( JTextArea output, String message )
    {
        outputArea = output;
        messageToOutput = message;
    }

    // method called to update outputArea
    public void run()
    {
        outputArea.append( messageToOutput );
    }
}
```
// Fig. 4.13: ProduceInteger.java
// Definition of threaded class ProduceInteger

// Java extension packages
import javax.swing.*;

public class ProduceInteger extends Thread {
    private HoldIntegerSynchronized sharedObject;
    private JTextArea outputArea;

    // initialize ProduceInteger
    public ProduceInteger( HoldIntegerSynchronized shared,
                           JTextArea output )
    {
        super( "ProduceInteger" );

        sharedObject = shared;
        outputArea = output;
    }

    // ProduceInteger thread loops 10 times and calls
    // sharedObject's setSharedInt method each time
    public void run()
    {
        for ( int count = 1; count <= 10; count++ ) {

            // sleep for a random interval
            // Note: Interval shortened purposely to fill buffer
            try {
                Thread.sleep( (int) ( Math.random() * 500 ) );
            }
        }
    }
}
// process InterruptedException during sleep
        catch( InterruptedException exception ) {
            System.err.println( exception.toString() );
        }

        sharedObject.setSharedInt( count );

        // update Swing GUI component
        SwingUtilities.invokeLater( new UpdateThread( outputArea,
            "\n" + getName() + " finished producing values" +
            "\nTerminating " + getName() + "\n" ) );

    } // end class ProduceInteger
// Fig. 4.14: ConsumeInteger.java
// Definition of threaded class ConsumeInteger

// Java extension packages
import javax.swing.*;

public class ConsumeInteger extends Thread {
    private HoldIntegerSynchronized sharedObject;
    private JTextArea outputArea;

    // initialize ConsumeInteger
    public ConsumeInteger( HoldIntegerSynchronized shared,
            JTextArea output )
    {
        super( "ConsumeInteger" );
        sharedObject = shared;
        outputArea = output;
    }
}
// ConsumeInteger thread loops until it receives 10
// from sharedObject's getSharedInt method
public void run()
{
    int value, sum = 0;

    do {
        // sleep for a random interval
        try {
            Thread.sleep( (int) ( Math.random() * 3000 ) );
        }

        // process InterruptedException during sleep
        catch( InterruptedException exception ) {
            System.err.println( exception.toString() );
        }

        value = sharedObject.getSharedInt();
        sum += value;
    } while ( value != 10 );

    // update Swing GUI component
    SwingUtilities.invokeLater( new UpdateThread( outputArea,
        "\n" + getName() + " retrieved values totaling: " +
        sum + "\nTerminating " + getName() + "\n" ) );
} // end class ConsumeInteger
// Fig. 4.15: HoldIntegerSynchronized.java
// Definition of class HoldIntegerSynchronized that
// uses thread synchronization to ensure that both
// threads access sharedInt at the proper times.

// Java core packages
import java.text.DecimalFormat;

// Java extension packages
import javax.swing.*;

public class HoldIntegerSynchronized {

    // array of shared locations
    private int sharedInt[] = { -1, -1, -1, -1, -1 };

    // variables to maintain buffer information
    private boolean writeable = true;
    private boolean readable = false;
    private int readLocation = 0, writeLocation = 0;

    // GUI component to display output
    private JTextArea outputArea;

    // initialize HoldIntegerSynchronized
    public HoldIntegerSynchronized( JTextArea output )
    {
        outputArea = output;
    }
}
// synchronized method allows only one thread at a time to
// invoke this method to set a value in a particular
// HoldIntegerSynchronized object
public synchronized void setSharedInt( int value )
{
    while ( !writeable ) {

        // thread that called this method must wait
        try {

            // update Swing GUI component
            SwingUtilities.invokeLater( new UpdateThread( outputArea, " WAITING TO PRODUCE " + value ) );

            wait();
        } 

        // process InterruptedException while thread waiting
        catch ( InterruptedException exception ) {
            System.err.println( exception.toString() );
        }
    }

    // place value in writeLocation
    sharedInt[ writeLocation ] = value;

    // indicate that consumer can read a value
    readable = true;

    // update Swing GUI component
    SwingUtilities.invokeLater( new UpdateThread( outputArea, "\nProduced " + value + " into cell " + writeLocation ) );
}
// update writeLocation for future write operation
writeLocation = (writeLocation + 1) % 5;

// update Swing GUI component
SwingUtilities.invokeLater(new UpdateThread(outputArea, "\twrite " + writeLocation + "\tread " + readLocation ));
displayBuffer(outputArea, sharedInt);

// test if buffer is full
if (writeLocation == readLocation) {
   writable = false;
   SwingUtilities.invokeLater(new UpdateThread(outputArea, "\nBUFFER FULL" ));
}

// tell a waiting thread to become ready
notify();
}

// synchronized method allows only one thread at a time to
// invoke this method to get a value from a particular
// HoldIntegerSynchronized object
public synchronized int getSharedInt()
{
    int value;
    while ( !readable ) {

// thread that called this method must wait
try {

   // update Swing GUI component
   SwingUtilities.invokeLater(new UpdateThread(
      outputArea, " WAITING TO CONSUME" ));

   wait();
}

// process InterruptedException while thread waiting
catch ( InterruptedException exception ) {
   System.err.println( exception.toString() );
}

// indicate that producer can write a value
writeable = true;

// obtain value at current readLocation
value = sharedInt[ readLocation ];

// update Swing GUI component
SwingUtilities.invokeLater(new UpdateThread(outputArea,
   "\nConsumed " + value + " from cell " + 
   readLocation ) );

// update read location for future read operation
readLocation = ( readLocation + 1 ) % 5;

// update Swing GUI component
SwingUtilities.invokeLater(new UpdateThread(
   outputArea, 
   "\nwrite " + writeLocation + "\tread " + 
   readLocation ) );
displayBuffer( outputArea, sharedInt );

// test if buffer is empty
if ( readLocation == writeLocation ) {
    readable = false;

    // update Swing GUI component
    SwingUtilities.invokeLater( new UpdateThread( outputArea, "\nBUFFER EMPTY" ) );
}

// tell a waiting thread to become ready
notify();

return value;
} // end method getSharedInt

// display contents of shared buffer
public void displayBuffer( JTextArea outputArea, int buffer[] )
{
    DecimalFormat formatNumber = new DecimalFormat( " #;-#" );
    StringBuffer outputBuffer = new StringBuffer();

    // place buffer elements in outputBuffer
    for ( int count = 0; count < buffer.length; count++ )
        outputBuffer.append( " " + formatNumber.format( buffer[ count ] ) );

    // update Swing GUI component
    SwingUtilities.invokeLater( new UpdateThread( outputArea, "\tbuffer: " + outputBuffer ) );
}
167  }  // end class HoldIntegerSynchronized
// Fig. 4.16: SharedCell.java
// Show multiple threads modifying shared object.

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.text.DecimalFormat;

// Java extension packages
import javax.swing.*;

public class SharedCell extends JFrame {

    // set up GUI
    public SharedCell() {
        super( "Demonstrating Thread Synchronization" );

        JTextArea outputArea = new JTextArea( 20, 30 );
        getContentPane().add( new JScrollPane( outputArea ) );
        setSize( 500, 500 );
        show();

        // set up threads
        HoldIntegerSynchronized sharedObject =
            new HoldIntegerSynchronized( outputArea );

        ProduceInteger producer =
            new ProduceInteger( sharedObject, outputArea );

        ConsumeInteger consumer =
            new ConsumeInteger( sharedObject, outputArea );
    }
}
35     // start threads
36     producer.start();
37     consumer.start();
38     }
39
40     // execute application
41     public static void main( String args[] )
42     {
43         SharedCell application = new SharedCell();
44         application.setDefaultCloseOperation(
45             JFrame.EXIT_ON_CLOSE);
46     }
47     }
48 } // end class SharedCell
Demonstrating Thread Synchronization

Produced 1 into cell 0  write 1  read 0  buffer: 1-1-1-1-1-1
Consumed 1 from cell 0  write 1  read 1  buffer: 1-1-1-1-1-1
BUFFER EMPTY
Produced 2 into cell 1  write 2  read 1  buffer: 1 2-1-1-1-1
Produced 3 into cell 2  write 3  read 1  buffer: 1 2 3-1-1-1
Produced 4 into cell 3  write 4  read 1  buffer: 1 2 3 4-1-1
Produced 5 into cell 4  write 0  read 1  buffer: 1 2 3 4 5
Produced 6 into cell 0  write 1  read 1  buffer: 6 2 3 4 5
BUFFER FULL WAITING TO PRODUCE 7
Consumed 2 from cell 1  write 1  read 2  buffer: 6 2 3 4 5
Produced 7 into cell 1  write 2  read 2  buffer: 6 7 3 4 5
BUFFER FULL WAITING TO PRODUCE 8
Consumed 3 from cell 2  write 2  read 3  buffer: 6 7 3 4 5
Produced 8 into cell 2  write 3  read 3  buffer: 6 7 8 4 5
BUFFER FULL WAITING TO PRODUCE 9
Consumed 4 from cell 3  write 3  read 4  buffer: 6 7 8 4 5
Produced 9 into cell 3  write 4  read 4  buffer: 6 7 8 9 5
BUFFER FULL WAITING TO PRODUCE 10
Consumed 5 from cell 4  write 4  read 0  buffer: 6 7 8 9 5
Produced 10 into cell 4  write 0  read 0  buffer: 6 7 8 9 10
BUFFER FULL
ProducenInteger finished producing values
Terminating ProducenInteger

Consumed 8 from cell 0  write 0  read 1  buffer: 6 7 8 9 10
Consumed 7 from cell 1  write 0  read 2  buffer: 6 7 8 9 10
Consumed 3 from cell 2  write 0  read 3  buffer: 6 7 8 9 10
Consumed 9 from cell 3  write 0  read 4  buffer: 6 7 8 9 10
Consumed 10 from cell 4  write 0  read 0  buffer: 6 7 8 9 10
BUFFER EMPTY
ConsumenInteger retrieved values totaling: 55
Terminating ConsumenInteger
4.9 Daemon Threads

- Runs for benefit of other threads
  - Do not prevent program from terminating
  - Garbage is a daemon thread

- Set daemon thread with method `setDaemon`
4.10 **Runnable Interface**

- Multithreading in a class that extends a class
  - A class cannot extend more than one class
  - Implements **Runnable** for multithreading support
- **Runnable** object grouped with a **Thread** object
// Fig. 4.17: RandomCharacters.java
// Demonstrating the Runnable interface.

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class RandomCharacters extends JApplet
    implements ActionListener {

    // declare variables used by applet and
    // inner class RunnableObject
    private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private final static int SIZE = 3;

    private JLabel outputs[];
    private JCheckBox checkboxes[];

    private Thread threads[];
    private boolean suspended[];

    // set up GUI and arrays
    public void init()
    {
        outputs = new JLabel[ SIZE ];
        checkboxes = new JCheckBox[ SIZE ];
        threads = new Thread[ SIZE ];
        suspended = new boolean[ SIZE ];

        Container container = getContentPane();
        container.setLayout( new GridLayout( SIZE, 2, 5, 5 ) );
    }

    String alphabet
    shared by three threads
// create GUI components, register listeners and attach
// components to content pane
for ( int count = 0; count < SIZE; count++ ) {
    outputs[ count ] = new JLabel();
    outputs[ count ].setBackground( Color.green );
    outputs[ count ].setOpaque( true );
    container.add( outputs[ count ] );
    checkboxes[ count ] = new JCheckBox( "Suspended" );
    checkboxes[ count ].addActionListener( this );
    container.add( checkboxes[ count ] );
}

// Create and start threads. This method called after init
// and when user revisits Web page containing this applet
public void start() {
    // create threads and start every time start is called
    for ( int count = 0; count < threads.length; count++ ) {
        // create Thread and initialize it with object that
        // implements Runnable
        threads[ count ] = new Thread( new RunnableObject(),
            "Thread " + ( count + 1 ) );

        // begin executing Thread
        threads[ count ].start();
    }
}
```java
private int getIndex( Thread current ) {
    for ( int count = 0; count < threads.length; count++ )
        if ( current == threads[ count ] )
            return count;
    return -1;
}

public synchronized void stop() {
    // Indicate that each thread should terminate. Setting
    // these references to null causes each thread's run
    // method to complete execution.
    for ( int count = 0; count < threads.length; count++ )
        threads[ count ] = null;

    // make all waiting threads ready to execute, so they
    // can terminate themselves
    notifyAll();
}
```
```java
// handle button events
public synchronized void actionPerformed( ActionEvent event )
{
    for ( int count = 0; count < checkboxes.length; count++ ) {
        if ( event.getSource() == checkboxes[ count ] ) {
            suspended[ count ] = !suspended[ count ];
            outputs[ count ].setBackground(
                !suspended[ count ] ? Color.green : Color.red );
            if ( !suspended[ count ] )
                notifyAll();
            return;
        }
    }
}

// private inner class that implements Runnable so objects
// of this class can control threads
private class RunnableObject implements Runnable {
    public void run()
    {
        // get reference to executing thread
        final Thread currentThread = Thread.currentThread();

        // determine thread's position in array
        final int index = getIndex( currentThread );
    }

    // Place random characters in GUI. Local variables
    // currentThread and index are declared final so
    // they can be used in an anonymous inner class.
    public void run()
    {
        // get reference to executing thread
        final Thread currentThread = Thread.currentThread();

        // determine thread's position in array
        final int index = getIndex( currentThread );
    }
```

Method `actionPerformed` detects clicking `Suspended` checkboxes.

Lines 93-111
- `Togge boolean` value in array `suspended`

Lines 101-102
- Set background color of the `JLabel` to red or green

Line 120
- Call `notifyAll` to start ready threads

Inner class `RunnableObject` controls threads.

Method `run` uses two local variables

Method `run` uses two local variables

Method `currentThread` gets reference to executing thread

Method `getIndex` determines index of currently running thread

All rights reserved.
```java
127 // loop condition determines when thread should stop
128 while ( threads[ index ] == currentThread ) {
129
130 // sleep from 0 to 1 second
131 try {
132     Thread.sleep( ( int ) ( Math.random() * 1000 ) );
133
134 // Determine whether thread should suspend
135 // execution. Use applet as monitor.
136 synchronized( RandomCharacters.this ) {
137
138     while ( suspended[ index ] &&
139         threads[ index ] == currentThread ) {
140
141         // Temporarily stop thread execution. Use
142         // applet as monitor.
143         RandomCharacters.this.wait();
144     }
145
146 } // end synchronized block
147
148 // process InterruptedExceptions during sleep or wait
149 catch ( InterruptedException interruptedException ) {
150     System.err.println( "sleep interrupted" );
151 }
152
153 }
```
### RandomCharacters.java

155  // display character on corresponding label
156  SwingUtilities.invokeLater(
157
158  // anonymous inner class used by SwingUtilities
159  // method invokeLater to ensure GUI
160  // updates properly
161  new Runnable() {
162
163  // updates Swing GUI component
164  public void run()
165  {
166  // pick random character
167  char displayChar = alphabet.charAt(
168  ( int ) ( Math.random() * 26 ) );
169
170  outputs[ index ].setText(
171  currentThread.getName() + ": " +
172  displayChar );
173
174  }
175  // end anonymous inner class
176
177  ); // end call to SwingUtilities.invokeLater
178
179  } // end while
180
181  System.err.println(
182  currentThread.getName() + " terminating" );
183
184  } // end private inner class RunnableObject
185
186  } // end class RandomCharacters
4.11 Thread Groups

• Class **ThreadGroup**
  – Create and manipulate groups of threads
  – Each group has unique name at creation

• Parent and child thread groups
  – Method calls sent to parent group also sent to child groups
4.12 (Optional Case Study) Thinking About Objects: Multithreading

- Concurrent models
  - UML contains support for building concurrent models
  - Discuss how simulation benefits from multithreading
    - `waitingPassenger` must wait for `ridingPassenger` to exit `Elevator`
    - Use `synchronized` method
      - Guarantees only one `Person` in `Elevator` at a time
4.12  (Optional Case Study) Thinking About Objects: Multithreading

• Threads, Active Classes and Synchronized method
  – UML represents a thread as an active class
    • Thick black border in UML diagram indicates an active class
  – Synchronized Methods in UML
    • Notation $\text{B/A}$ to indicates $\text{A}$ must wait for $\text{B}$ to finish
4.12  (Optional Case Study) Thinking About Objects: Multithreading

• **Sequence Diagrams**
  – Shows interactions between objects
    • Shows messages passed between objects over time
  – Rectangle enclosing an object name represents that object
    • Use same naming conventions as collaboration diagrams
  – **Lifeline**
    • Dotted line running down from an object name
    • Actions occur on lifeline in chronological order, top to bottom
  – **Arrows**
    • Dashed arrows
      – Represent “return messages,” return of control to object
    • Solid arrows
      – A message sent from one object to another
4.12  (Optional Case Study) Thinking About Objects: Multithreading

• Class diagram now uses active classes
  – Elevator and Person are now active classes
4.12 (Optional Case Study) Thinking About Objects: Multithreading

Fig. 4.18 Modified collaboration diagram with active classes for passengers entering and exiting the Elevator.
4.12 (Optional Case Study) Thinking About Objects: Multithreading

Fig. 4.19  Sequence diagram for a single Person changing floors in system.
4.12 (Optional Case Study) Thinking About Objects: Multithreading

Fig. 4.20 Final class diagram of the elevator simulation
4.13 (Optional) Discovering Design Patterns: Concurrent Design Patterns

• Concurrency Design Patterns
  – Single-Threaded Execution design pattern
    • Stops several threads from invoking a method concurrently
  – Guarded Suspension design pattern
    • Suspends and resumes a thread's activity when a condition met
  – Balking design pattern
    • Causes method to balk if an object occupies a certain state
  – Read/Write Lock design pattern
    • Multiple read on an object but exclusive write
  – Two-Phase Termination design pattern
    • Uses two-phase termination process to ensure resources freed
4.13 (Optional) Discovering Design Patterns: Concurrent Design Patterns

Fig. 4.21  Final class diagram with attributes and operations.

<table>
<thead>
<tr>
<th>ElevatorModel</th>
<th>Light</th>
</tr>
</thead>
<tbody>
<tr>
<td>- numberOfPeople : Integer = 0</td>
<td></td>
</tr>
<tr>
<td>+ addPerson() : void</td>
<td></td>
</tr>
<tr>
<td>- lightOn : Boolean = false</td>
<td></td>
</tr>
<tr>
<td>+ turnOnLight() : void</td>
<td></td>
</tr>
<tr>
<td>+ turnOffLight() : void</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>ElevatorShaft</th>
</tr>
</thead>
<tbody>
<tr>
<td>- locationName : String</td>
<td></td>
</tr>
<tr>
<td># setLocationName(String) : void</td>
<td></td>
</tr>
<tr>
<td>+ getLocationName() : String</td>
<td></td>
</tr>
<tr>
<td>+ getButton() : Button</td>
<td></td>
</tr>
<tr>
<td>+ getDoor() : Door</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person</th>
<th>Floor</th>
</tr>
</thead>
<tbody>
<tr>
<td>- ID : Integer</td>
<td></td>
</tr>
<tr>
<td>- moving : Boolean = true</td>
<td></td>
</tr>
<tr>
<td>- location : Location</td>
<td></td>
</tr>
<tr>
<td>- maxTravelTime : Integer = 10 * 60</td>
<td></td>
</tr>
<tr>
<td>+ doorOpen() : void</td>
<td></td>
</tr>
<tr>
<td>+ getButton() : Button</td>
<td></td>
</tr>
<tr>
<td>+ getDoor() : Door</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Elevator</th>
<th>Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>- moving : Boolean = false</td>
<td></td>
</tr>
<tr>
<td>- summoned : Boolean = false</td>
<td></td>
</tr>
<tr>
<td>- currentFloor : Location</td>
<td></td>
</tr>
<tr>
<td>- destinationFloor : Location</td>
<td></td>
</tr>
<tr>
<td>- travelTime : Integer = 5</td>
<td></td>
</tr>
<tr>
<td>+ ride() : void</td>
<td></td>
</tr>
<tr>
<td>+ requestElevator() : void</td>
<td></td>
</tr>
<tr>
<td>+ setMoving(Boolean) : void</td>
<td></td>
</tr>
<tr>
<td>+ getButton() : Button</td>
<td></td>
</tr>
<tr>
<td>+ getDoor() : Door</td>
<td></td>
</tr>
<tr>
<td>- pressed : Boolean = false</td>
<td></td>
</tr>
<tr>
<td>+ resetButton() : void</td>
<td></td>
</tr>
<tr>
<td>+ pressButton() : void</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Door</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- open : Boolean = false</td>
<td></td>
</tr>
<tr>
<td>+ openDoor() : void</td>
<td></td>
</tr>
<tr>
<td>+ closeDoor() : void</td>
<td></td>
</tr>
</tbody>
</table>