Software Design 2
SDN260S

Multithreading

H. Mataifa

Department of Electronic, Electrical and Computer Engineering

Outline

- Background: what threads are
- Thread states and life cycle
- Creating and executing threads
- Thread synchronizaton
- Producer/Consumer relationships

Background: Multithreading

- **Concurrency in computers**:
  - Computers normally perform several tasks in parallel/concurrently, e.g. compile a program, send a file to a printer, receive email over a network, etc.
  - **Concurrency**: performing more than one task at the same time (using a single processor or multiple processors)
  - Concurrency is usually achieved by means of threads
  - **Thread**: a *path of execution within a process* (or, a single sequential flow of control within a program)
  - A program/process can have more than one thread
- **Multithreading**:
  - A program with multiple threads of execution, where each thread has its own method-call stack and program counter, enabling it to execute concurrently with other threads while sharing with them application-wide resources, e.g. memory
  - Enables tasks to be distributed across multiple processors (if available), increasing program responsiveness and efficiency (better use of computing resources)

Background: Multithreading

- **Concurrent programming**:

  - A concurrent program is essentially a multithreaded program; it contains two or more threads that execute concurrently and work together to perform some task

- **Challenges of multithreaded/concurrent programming**:

  - Has to take care of issues such as synchronization, race conditions, deadlock handling, etc.

- **Support for Concurrent Programming in Java**:

  - To avoid many difficulties and errors associated with concurrent programs, recommended to use Java's *prebuilt concurrency capabilities* (Concurrency **API**s) to write multithreaded programs

  - "Simple" is better in multithreaded programming; use advanced **API**s (e.g. **Lock, Condition**) only when dictated by application requirements

Thread States: Life Cycle of a Thread

- **Thread states**:
  - A new thread will be in any one of the states: (1) *runnable,* (2) *waiting,* (3) *timed waiting,* (4) *blocked,* (5) *terminated*
  - Various conditions apply that will transition the thread from one state to another
  - Final state is *terminated*, when the thread has successfully completed its designated task, or otherwise terminates due to an error

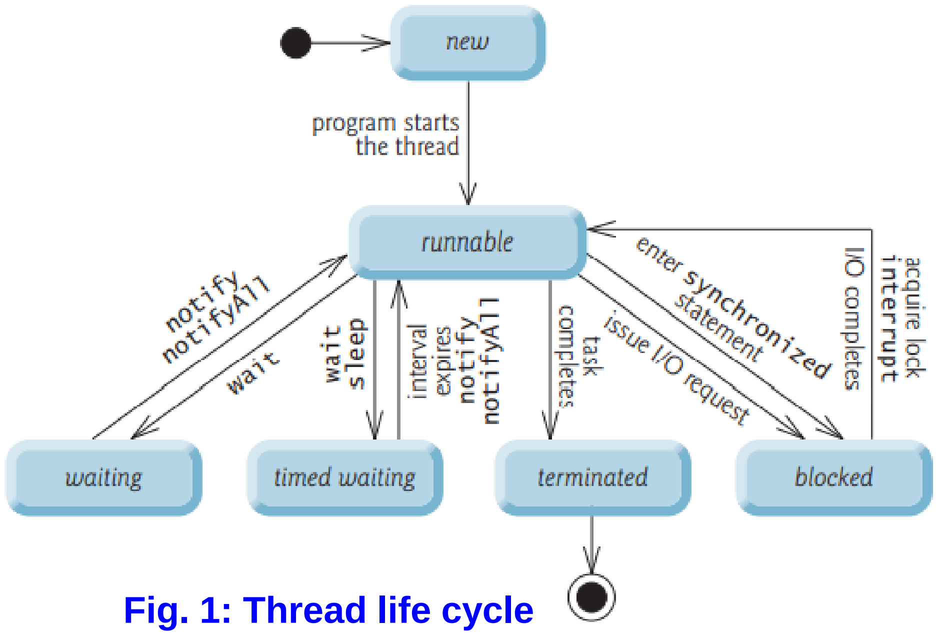
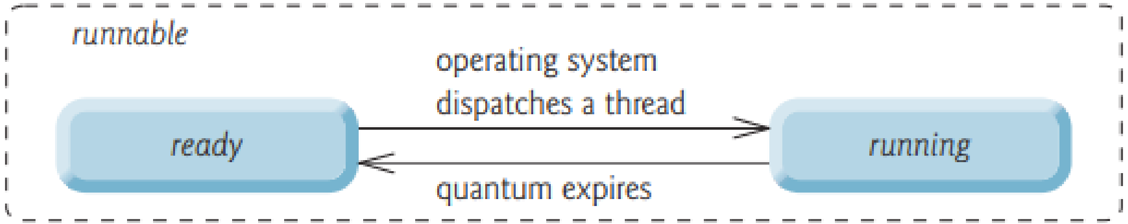


Fig. 1: Thread life cycle

Waitinfg for a resoiurce that is being used by anoher thread

Thread States: Life Cycle of a Thread

- **Thread states: Runnable state**:
  - At the operating-system level, thread's runnable state typically has two separate states: *ready* and *running*
  - *Ready* state can be seen as the entry to *runnable* state (intermediate state before the thread can actually get to run)
  - A *ready* thread enters the *running* state (i.e. begins executing) when the operating system assigns it to a processor- also known as *dispatching the thread*
  - **Thread scheduling**: the process an operating system uses to determine which thread to dispatch
  - The operating system uses a thread priority (a property of every Java thread) in thread scheduling; higher-priority threads have preference in processor allocation
  - Typically, each thread is given a quantum or timeslice in which to perform a task



Thread is ready to run but it wis now running becasue it is waiting to be dispathed

D

Fig. 2: Thread's runnable state

Thread States: Life Cycle of a Thread

- **Thread scheduling**:
  - An operating-system's thread scheduler determines which thread runs next
  - Most operating systems support timeslicing, which enables threads of equal priority to share a processor, in a round-robin fashion, until all threads run to completion
  - **Preemptive scheduling**: when a higher-priority thread enters the ready state, the operating system preempts the currently running thread (essentially reassigns the processor to the higher-priority thread)
  - **Indefinite postponement**/**starvation**: could occur when higher-priority threads keep postponing the execution of lower-priority threads
  - **Aging**: employed by the operating system to prevent starvation (essentially, a thread's priority increases with the amount of time it's been waiting)
  - Thread scheduling is platform-dependent: the behaviour of a multithreaded program could vary across different Java implementations
  - Java's **Concurrency APIs** hide much of the issues associated with Thread Management

Creating and Executing Threads with Executor Framework

- **Interface Runnable**:
  - A class implements the **Runnable** interface to be able to perform a task that can execute concurrently with other tasks
  - Interface Runnable declares the single method **run**, which contains the code that defines the task that a Runnable object should perform
  - When a thread executing a Runnable object is created and started, the thread calls the Runnable object's run method, which executes the task in the new thread

- **Executor:**
  - Responsible for creating and managing a group of threads (thread pool)
  - **Executor** method **execute** accepts a Runnable object as argument, which **Executor** assigns to one of the available threads in a thread pool
  - **Executor framework** provides a means for efficient thread management; recommended over custom thread creation and management

Creating and Executing Threads with Executor Framework

- **Interface ExecutorService**:
  - Extends **Executor** interface, declares various methods for managing the life cycle of an Executor
  - Uses Class **Executors** method newCachedThreadPool to create a thread pool

- **Class PrintTask:**
  - Implements **Runnable** to enable concurrent execution of PrintTasks
  - **Thread** static method sleep places a thread in the *timed waiting* state for a specified amount of time
  - The program has **main** thread (created by **JVM**) and other threads created in main, which execute the PrintTasks
  - All threads (main as well as those created within main) should terminate for the whole program to terminate
  - **ExecutorService** method shutdown signals termination of the thread pool; tasks already submitted are executed to completion, but no new tasks are accepted

Class PrintTask

```java
// Fig. 26.3: PrintTask.java
// PrintTask class sleeps for a random time from 0 to 5 seconds
import java.util.Random;

public class PrintTask implements Runnable
{
   private final int sleepTime; // random sleep time for thread
   private final String taskName; // name of task
   private final static Random generator = new Random();

   // constructor
   public PrintTask( String name )
   {
      taskName = name; // set task name

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

   // method run contains the code that a thread will execute
   public void run()
   {
      try // put thread to sleep for sleepTime amount of time
      {
         System.out.printf( "%s going to sleep for %d milliseconds.\n",
            taskName, sleepTime );
         Thread.sleep( sleepTime ); // put thread to sleep
      } // end try
      catch ( InterruptedException exception )
      {
         System.out.printf( "%s %s\n", taskName,
            "terminated prematurely due to interruption" );
      } // end catch

      // print task name
      System.out.printf( "%s done sleeping\n", taskName );
   } // end method run
} // end class PrintTask
```

Class TaskExecutor

```java
// Fig. 26.4: TaskExecutor.java
// Using an ExecutorService to execute Runnables.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

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

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

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

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

      // shut down worker threads when their tasks complete
      threadExecutor.shutdown();

      System.out.println( "Tasks started, main ends.\n" );
   } // end main
} // end class TaskExecutor
```

Threads are created here

when we do this we are freeing the memory

```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
task3 done sleeping
task2 done sleeping
task1 done sleeping
```

```
Starting Executor
task1 going to sleep for 3161 milliseconds.
task3 going to sleep for 532 milliseconds.
task2 going to sleep for 3440 milliseconds.
Tasks started, main ends.

task3 done sleeping
task1 done sleeping
task2 done sleeping
```

Thread Synchronization

- **Thread synchronization**:
  - Coordinated access to shared data by multiple concurrent threads
  - Ensures that only one thread has exclusive access to a shared object that can be modified by all threads
  - Prevents indeterminate results that may occur if access to shared object is not controlled
  - When one thread has exclusive access to the shared object, other threads have to wait before getting access to the object
  - **Mutual exclusion**: ensures that each thread accessing a shared object excludes all others from doing so simultaneously

- **Synchronization via Monitors:**
  - Every object has a monitor and a monitor lock (or intrinsic lock) which can be held by only one thread at a time; can be used to gain exclusive access to the object
  - To modify the object, a thread needs to acquire the lock; other threads attempting to modify the object are *blocked* until the lock is released
  - A synchronized statement is used to indicate that lock acquisition is needed to access the object; the object is said to be guarded by a monitor lock

Thread Synchronization

- **Synchronized statement**:

```
synchronized ( object )
{
   statements
} // end synchronized statement
```

  - object is the object whose monitor lock needs to be acquired in order to modify it; it is normally this if it's the object in which the synchronized statement appears
  - The monitor lock is automatically released once the synchronized statement is exited
  - Java also allows synchronized methods (a monitor lock must be acquired for the method to be invoked on an object)

- **Unsynchronized data sharing (Programs 26.5-26.7):**
  - **SimpleArray** object (Prog. 26.5) is shared among multiple threads, which can all place elements into the array; **Thread** method sleep used to highlight issues associated with unsynchronized access to the shared data
  - Class **ArrayWriter** (prog. 26.6) implements interface **Runnable** to define a task for inserting elements into **SimpleArray**; task completes after placing three consecutive values into the array
  - Class **SharedArrayTest** (prog. 26.7) executes two **ArrayWriter** tasks that add values to a single **SimpleArray** object
  - **ExecutorService** method awaitTermination returns control to its caller either when all tasks complete or when the specified timeout elapses

Unsynchronized Data Sharing: SimpleArray

```java
// Fig. 26.5: SimpleArray.java
// Class that manages an integer array to be shared by multiple threads.
import java.util.Arrays;
import java.util.Random;

public class SimpleArray // CAUTION: NOT THREAD SAFE!
{
   private final int[] array; // the shared integer array
   private int writeIndex = 0; // index of next element to be written
   private final static Random generator = new Random();

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

   // add a value to the shared array
   public void add( int value )
   {
      int position = writeIndex; // store the write index

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

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

      ++writeIndex; // increment index of element to be written next
      System.out.printf( "Next write index: %d\n", writeIndex );
   } // end method add

   // used for outputting the contents of the shared integer array
   public String toString()
   {
      return "\nContents of SimpleArray:\n" + Arrays.toString( array );
   } // end method toString
} // end class SimpleArray
```

Unsynchronized Data Sharing: ArrayWriter

```java
// Fig. 26.6: ArrayWriter.java
// Adds integers to an array shared with other Runnables
import java.lang.Runnable;

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

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

   public void run()
   {
      for ( int i = startValue; i < startValue + 3; i++ )
      {
         sharedSimpleArray.add( i ); // add an element to the shared array
      } // end for
   } // end method run
} // end class ArrayWriter
```

This is to show that there are multiple threads writing and based on the sequence that will determine the output

Unsynchronized Data Sharing: SharedArrayTest

```java
// Fig 26.7: SharedArrayTest.java
// Executes two Runnables to add elements to a shared SimpleArray.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

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

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

      // execute the tasks with an ExecutorService
      ExecutorService executor = Executors.newCachedThreadPool();
      executor.execute( writer1 );
      executor.execute( writer2 );

      executor.shutdown();

      try
      {
         // wait 1 minute for both writers to finish executing
         boolean tasksEnded = executor.awaitTermination(
            1, TimeUnit.MINUTES );

         if ( tasksEnded )
            System.out.println( sharedSimpleArray ); // print contents
         else
            System.out.println(
               "Timed out while waiting for tasks to finish." );
      } // end try
      catch ( InterruptedException ex )
      {
         System.out.println(
            "Interrupted while waiting for tasks to finish." );
      } // end catch
   } // end main
} // end class SharedArrayTest
```

By default we do not specify how many
threads we will use we can only specify how man task

Dispathc the thread and pass
it to the cpu

We use a try block to check if the the thread is availabe

Catch any thread that is not available

Unsynchronized Data Sharing: SharedArrayTest

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

Contents of SimpleArray:
[11, 2, 3, 0, 12, 13]
```

First `pool-1-thread-1` wrote the value `1` to element `0`. Later `pool-1-thread-2` wrote the value `11` to element `0`, thus *overwriting* the previously stored value.

Note:

➢ One of the challenges of multithreaded programming is the spotting of errors – they may occur so infrequently that a ‘broken’ program does not produce incorrect results during testing, creating the illusion that the program is correct

Synchronized Data Sharing: Making Operations Atomic

- **Thread safety**:
  - An object is thread-safe if it can be accessed by multiple threads concurrently without any errors occurring; SimpleArray is **not** thread-safe
  - **SimpleArray** is not thread-safe because it allows any number of threads to read and modify shared data concurrently, which tends to cause errors
  - To make **SimpleArray** thread-safe, method **add** is turned into a synchronized method, then at any one time, only one thread can modify **SimpleArray**
  - **Atomic operation**: a set of operations that cannot be divided into smaller sub-operations; they are placed in a synchronized statement to make them atomic
  - In a multithreaded program, all accesses to mutable data shared by multiple threads are placed in synchronized statements or synchronized methods

Class SimpleArray with Synchronization

What ever thread is going to write or read thread they will need to use the add method which is synchronized

```
// Fig. 26.8: SimpleArray.java
// Class that manages an integer array to be shared by multiple
// threads with synchronization.
import java.util.Arrays;
import java.util.Random;

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

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

   // add a value to the shared array
   public synchronized void add( int value )
   {
      int position = writeIndex; // store the write index

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

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

      ++writeIndex; // increment index of element to be written next
      System.out.printf( "Next write index: %d\n", writeIndex );
   } // end method add

   // used for outputting the contents of the shared integer array
   public String toString()
   {
      return "\nContents of SimpleArray:\n" + Arrays.toString( array );
   } // end method toString
} // end class SimpleArray
```

the synchronised keyword is used to obtain a lock to the buffer

This keyword is used to achieve multi exclusion

Class SimpleArray with Synchronization

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

Contents of SimpleArray:
1 11 12 13 2 3
```

Notes:

- Avoid performing **I/O** operations (or lengthy calculations not requiring synchronization) in synchronized blocks; it's important to minimize amount of time that an object is "locked"
- Avoid calling Thread method "sleep" while holding a lock
- Keep duration of synchronized statements as short as possible while maintaining the needed synchronization; this minimizes wait time for blocked threads
- Declare data fields that are not expected to change as *final* (can be primitives or object references)

Producer/Consumer Relationship

- **Producer/consumer relationship**:
  - In a producer/consumer relationship, Producer thread generates data and stores it in a shared object, Consumer thread reads data from the shared object
  - Shared object is called a buffer
    - Producer writes to buffer
    - Consumer reads from buffer
  - Producer/consumer relationship requires synchronization to ensure values are produced and consumed properly
  - Operations on buffer data shared by producer/consumer are state-dependent; buffer has to be in correct state for each operation
  - Producer may write to buffer if in not-full state; Consumer may read from buffer if in not-empty state
  - **Programs 26.9 – 26.13**: details an example of a Producer/Consumer relationship without synchronization

Producer/Consumer Relationship: Buffer Interface

```java
// Fig. 26.9: Buffer.java
// Buffer interface specifies methods called by Producer and Consumer.
public interface Buffer
{
   // place int value into Buffer
   public void set( int value ) throws InterruptedException;

   // return int value from Buffer
   public int get() throws InterruptedException;
} // end interface Buffer
```

Producer/Consumer Relationship: Producer Thread

```java
1  // Fig. 26.10: Producer.java
2  // Producer with a run method that inserts the values 1 to 10 in buffer.
3  import java.util.Random;
4
5  public class Producer implements Runnable
6  {
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Producer( Buffer shared )
12    {
13        sharedLocation = shared;
14    } // end Producer constructor
15
16    // store values from 1 to 10 in sharedLocation
17    public void run()
18    {
19       int sum = 0;
20
21       for ( int count = 1; count <= 10; count++ )
22       {
23          try // sleep 0 to 3 seconds, then place value in Buffer
24          {
25             Thread.sleep( generator.nextInt( 3000 ) ); // random sleep
26             sharedLocation.set( count ); // set value in buffer
27             sum += count; // increment sum of values
28             System.out.printf( "\t%2d\n", sum );
29          } // end try
30          // if lines 25 or 26 get interrupted, print stack trace
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // end catch
35       } // end for
36
37       System.out.println(
38          "Producer done producing\nTerminating Producer" );
39    } // end method run
40 } // end class Producer
```

Producer/Consumer Relationship: Consumer Thread

```java
// Fig. 26.11: Consumer.java
// Consumer with a run method that loops, reading 10 values from buffer.
import java.util.Random;

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

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

   // read sharedLocation's value 10 times and sum the values
   public void run()
   {
      int sum = 0;

      for ( int count = 1; count <= 10; count++ )
      {
         // sleep 0 to 3 seconds, read value from buffer and add to sum
         try
         {
            Thread.sleep( generator.nextInt( 3000 ) );
            sum += sharedLocation.get();
            System.out.printf( "\t\t\t%2d\n", sum );
         } // end try
         // if lines 26 or 27 get interrupted, print stack trace
         catch ( InterruptedException exception )
         {
            exception.printStackTrace();
         } // end catch
      } // end for

      System.out.printf( "\n%s %d\n%s\n",
         "Consumer read values totaling", sum, "Terminating Consumer" );
   } // end method run
} // end class Consumer
```

Producer/Consumer Relationship: Unsynchronized Buffer

```java
// Fig. 26.12: UnsynchronizedBuffer.java
// UnsynchronizedBuffer maintains the shared integer that is accessed by
// a producer thread and a consumer thread via methods set and get.
public class UnsynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consumer threads

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

   // return value from buffer
   public int get() throws InterruptedException
   {
      System.out.printf( "Consumer reads\t%2d", buffer );
      return buffer;
   } // end method get
} // end class UnsynchronizedBuffer
```

Producer/Consumer Relationship: SharedBufferTest

```java
// Fig. 26.13: SharedBufferTest.java
// Application with two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

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

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

      System.out.println(
         "Action\t\tValue\tSum of Produced\tSum of Consumed" );
      System.out.println(
         "------\t\t-----\t---------------\t---------------\n" );

      // execute the Producer and Consumer, giving each of them access
      // to sharedLocation
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown(); // terminate application when tasks complete
   } // end main
} // end class SharedBufferTest
```

Producer/Consumer Relationship: SharedBufferTest

```
Action           Value    Sum of Produced Sum of Consumed
------           -----    --------------- ---------------

Producer writes  1          1
Producer writes  2          3                                 —— 1 is lost
Producer writes  3          6                                 —— 2 is lost
Consumer reads   3                           3
Producer writes  4         10
Consumer reads   4                           7
Producer writes  5         15
Producer writes  6         21                                 —— 5 is lost
Producer writes  7         28                                 —— 6 is lost
Consumer reads   7                          14
Consumer reads   7                          21                —— 7 read again
Producer writes  8         36
Consumer reads   8                          29
Consumer reads   8                          37                —— 8 read again
Producer writes  9         45
Producer writes 10         55                                 —— 9 is lost

Producer done producing
Terminating Producer
Consumer reads  10                          47
Consumer reads  10                          57                —— 10 read again
Consumer reads  10                          67                —— 10 read again
Consumer reads  10                          77                —— 10 read again

Consumer read values totaling 77
Terminating Consumer
```

Producer/Consumer Relationship: ArrayBlockingQueue

- **Class ArrayBlockingQueue**:
  - A fully implemented, thread-safe buffer class that implements interface **BlockingQueue**
    - Declares methods put and take, the blocking equivalents of **Queue** methods offer and poll respectively
  - Method put places elements at the end of the **BlockingQueue**, waiting if the queue is full
  - Method take removes an element from the head of the **BlockingQueue**, waiting if the queue is empty

Producer/Consumer Relationship: BlockingBuffer

```java
// Fig. 26.14: BlockingBuffer.java
// Creating a synchronized buffer using an ArrayBlockingQueue.
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingBuffer implements Buffer
{
   private final ArrayBlockingQueue<Integer> buffer; // shared buffer

   public BlockingBuffer()
   {
      buffer = new ArrayBlockingQueue<Integer>( 1 );
   } // end BlockingBuffer constructor

   // place value into buffer
   public void set( int value ) throws InterruptedException
   {
      buffer.put( value ); // place value in buffer

      System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
         "Buffer cells occupied: ", buffer.size() );
   } // end method set

   // return value from buffer
   public int get() throws InterruptedException
   {
      int readValue = buffer.take(); // remove value from buffer
      System.out.printf( "%s %2d\t%s%d\n", "Consumer reads ",
         readValue, "Buffer cells occupied: ", buffer.size() );

      return readValue;
   } // end method get
} // end class BlockingBuffer
```

Producer/Consumer Relationship: BlockingBufferTest

```java
// Fig. 26.15: BlockingBufferTest.java
// Two threads manipulating a blocking buffer that properly
// implements the producer/consumer relationship.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

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

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

      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class BlockingBufferTest
```

Producer/Consumer Relationship: BlockingBufferTest

```
Producer writes  1      Buffer cells occupied: 1
Consumer reads   1      Buffer cells occupied: 0
Producer writes  2      Buffer cells occupied: 1
Consumer reads   2      Buffer cells occupied: 0
Producer writes  3      Buffer cells occupied: 1
Consumer reads   3      Buffer cells occupied: 0
Producer writes  4      Buffer cells occupied: 1
Consumer reads   4      Buffer cells occupied: 0
Producer writes  5      Buffer cells occupied: 1
Consumer reads   5      Buffer cells occupied: 0
Producer writes  6      Buffer cells occupied: 1
Consumer reads   6      Buffer cells occupied: 0
Producer writes  7      Buffer cells occupied: 1
Consumer reads   7      Buffer cells occupied: 0
Producer writes  8      Buffer cells occupied: 1
Consumer reads   8      Buffer cells occupied: 0
Producer writes  9      Buffer cells occupied: 1
Consumer reads   9      Buffer cells occupied: 0
Producer writes 10      Buffer cells occupied: 1

Producer done producing
Terminating Producer
Consumer reads  10      Buffer cells occupied: 0

Consumer read values totaling 55
Terminating Consumer
```

Producer/Consumer Relationship with Synchronization

- **Custom shared buffer with synchronization**:
  - Use of synchronized keyword and methods of class **Object**
  - **Buffer** methods get and set are implemented as synchronized methods; a thread must obtain a monitor lock for the buffer to be able to operate on it
  - Depending on whether buffer is in correct state or not, Object methods wait, notify, notifyAll are used to transition threads between runnable and waiting states
  - Object method wait places calling thread in waiting state and releases monitor lock on the buffer, enabling any thread in runnable state to acquire the lock
  - Object method notify allows a thread in waiting state to transition back to runnable state; method notifyAll allows all threads in waiting state to become runnable
  - **IllegalMonitorStateException** results from a thread issuing wait, notify, notifyAll signals on an object without having a monitor lock to the object
  - Using notifyAll ensures that all waiting threads become runnable eventually, to prevent thread starvation

Producer/Consumer Relationship Synchronized Buffer

```java
// Fig. 26.16: SynchronizedBuffer.java
// Synchronizing access to shared data using Object
// methods wait and notifyAll.
public class SynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consumer threads
   private boolean occupied = false; // whether the buffer is occupied

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

      buffer = value; // set new buffer value

      // indicate producer cannot store another value
      // until consumer retrieves current buffer value
      occupied = true;

      displayState( "Producer writes " + buffer );

      notifyAll(); // tell waiting thread(s) to enter runnable state
   } // end method set; releases lock on SynchronizedBuffer

```

Producer/Consumer Relationship Synchronized Buffer

```java
   // return value from buffer
   public synchronized int get() throws InterruptedException
   {
      // while no data to read, place thread in waiting state
      while ( !occupied )
      {
         // output thread information and buffer information, then wait
         System.out.println( "Consumer tries to read." );
         displayState( "Buffer empty. Consumer waits." );
         wait();
      } // end while

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

      displayState( "Consumer reads " + buffer );

      notifyAll(); // tell waiting thread(s) to enter runnable state

      return buffer;
   } // end method get; releases lock on SynchronizedBuffer

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

Producer/Consumer Relationship SharedBufferTest

```java
// Fig. 26.17: SharedBufferTest2.java
// Two threads correctly manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest2
{
   public static void main( String[] args )
   {
      // create a newCachedThreadPool
      ExecutorService application = Executors.newCachedThreadPool();

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

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

      // execute the Producer and Consumer tasks
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class SharedBufferTest2
```

Producer/Consumer Relationship SharedBufferTest

Operation	Buffer	Occupied
Consumer tries to read. Buffer empty. Consumer waits.	-1	false
Producer writes 1	1	true
Consumer reads 1	1	false
Consumer tries to read. Buffer empty. Consumer waits.	1	false
Producer writes 2	2	true
Consumer reads 2	2	false
Producer writes 3	3	true
Consumer reads 3	3	false
Producer writes 4	4	true
Producer tries to write. Buffer full. Producer waits.	4	true
Consumer reads 4	4	false
Producer writes 5	5	true
Consumer reads 5	5	false
Producer writes 6	6	true
Producer tries to write. Buffer full. Producer waits.	6	true
Consumer reads 6	6	false
Producer writes 7	7	true
Producer tries to write. Buffer full. Producer waits.	7	true
Consumer reads 7	7	false
Producer writes 8	8	true
Consumer reads 8	8	false
Consumer tries to read. Buffer empty. Consumer waits.	8	false

Producer/Consumer Relationship SharedBufferTest

```
Producer writes 9                   9           true

Consumer reads 9                    9           false

Consumer tries to read.
Buffer empty. Consumer waits.       9           false

Producer writes 10                  10          true

Consumer reads 10                   10          false


Producer done producing
Terminating Producer

Consumer read values totaling 55
Terminating Consumer
```

Producer/Consumer Relationship Bounded Buffers

- **Problem with producer/consumer program in section 26.7**:

We want the buffer to be of right size

  - Application's performance significantly affected by relative speeds of Producer and Consumer threads
  - If Producer thread produces much faster than Consumer thread can consume, Producer has to wait long until buffer has space to place next value
  - When threads wait excessively, programs become less efficient, interactive programs become less responsive, applications suffer longer delays

- **Solution: bounded buffer**:
  - Provides a fixed number of buffer cells into which the Producer thread can place values, and from which the Consumer thread can retrieve the values
  - Minimizes the amount of waiting time for the threads sharing the buffer resource
  - Buffer size should be optimized to minimize amount of thread wait time, while not wasting space

- **CircularBuffer:**
  - Writes into and reads from the array elements in order, beginning at the first cell and moving towards the last
  - When a Producer/Consumer reaches the last element, it returns to the first and begins writing/reading there

We basically want to reduce the program from consuming even thoght the producer is producing using a buffer will help to stop this

Producer/Consumer Relationship CircularBuffer

```java
// Fig. 26.18: CircularBuffer.java
// Synchronizing access to a shared three-element bounded buffer.
public class CircularBuffer implements Buffer
{
   private final int[] buffer = { -1, -1, -1 }; // shared buffer

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

   // place value into buffer
   public synchronized void set( int value ) throws InterruptedException
   {
      // wait until buffer has space available, then write value;
      // while no empty locations, place thread in blocked state
      while ( occupiedCells == buffer.length )
      {
         System.out.printf( "Buffer is full. Producer waits.\n" );
         wait(); // wait until a buffer cell is free
      } // end while

      buffer[ writeIndex ] = value; // set new buffer value

      // update circular write index
      writeIndex = ( writeIndex + 1 ) % buffer.length;

      ++occupiedCells; // one more buffer cell is full
      displayState( "Producer writes " + value );
      notifyAll(); // notify threads waiting to read from buffer
   } // end method set

   // return value from buffer
   public synchronized int get() throws InterruptedException
   {
      // wait until buffer has data, then read value;
      // while no data to read, place thread in waiting state
      while ( occupiedCells == 0 )
      {
         System.out.printf( "Buffer is empty. Consumer waits.\n" );
```

Producer/Consumer Relationship CircularBuffer

```java
         wait(); // wait until a buffer cell is filled
      } // end while

      int readValue = buffer[ readIndex ]; // read value from buffer

      // update circular read index
      readIndex = ( readIndex + 1 ) % buffer.length;

      --occupiedCells; // one fewer buffer cells are occupied
      displayState( "Consumer reads " + readValue );
      notifyAll(); // notify threads waiting to write to buffer

      return readValue;
   } // end method get

   // display current operation and buffer state
   public void displayState( String operation )
   {
      // output operation and number of occupied buffer cells
      System.out.printf( "%s%s%d)\n%s", operation,
         " (buffer cells occupied: ", occupiedCells, "buffer cells:  " );

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

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

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

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

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

      System.out.println( "\n" );
   } // end method displayState
} // end class CircularBuffer
```

Producer/Consumer Relationship CircularBufferTest

```java
// Fig. 26.19: CircularBufferTest.java
// Producer and Consumer threads manipulating a circular buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

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

      // create CircularBuffer to store ints
      CircularBuffer sharedLocation = new CircularBuffer();

      // display the initial state of the CircularBuffer
      sharedLocation.displayState( "Initial State" );

      // execute the Producer and Consumer tasks
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class CircularBufferTest
```

Producer/Consumer Relationship CircularBufferTest

```
Initial State (buffer cells occupied: 0)
buffer cells:   -1   -1   -1
               ---- ---- ----
                WR

Producer writes 1 (buffer cells occupied: 1)
buffer cells:    1   -1   -1
               ---- ---- ----
                 R    W
```

```
Consumer reads 1 (buffer cells occupied: 0)
buffer cells:    1   -1   -1
               ---- ---- ----
                     WR

Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
buffer cells:    1    2   -1
               ---- ---- ----
                      R    W

Consumer reads 2 (buffer cells occupied: 0)
buffer cells:    1    2   -1
               ---- ---- ----
                          WR

Producer writes 3 (buffer cells occupied: 1)
buffer cells:    1    2    3
               ---- ---- ----
                W          R

Consumer reads 3 (buffer cells occupied: 0)
buffer cells:    1    2    3
               ---- ---- ----
                WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells:    4    2    3
               ---- ---- ----
                 R    W

Producer writes 5 (buffer cells occupied: 2)
buffer cells:    4    5    3
               ---- ---- ----
                 R         W

Consumer reads 4 (buffer cells occupied: 1)
buffer cells:    4    5    3
               ---- ---- ----
                      R    W

Producer writes 6 (buffer cells occupied: 2)
buffer cells:    4    5    6
               ---- ---- ----
                W     R

Producer writes 7 (buffer cells occupied: 3)
buffer cells:    7    5    6
               ---- ---- ----
                     WR

Consumer reads 5 (buffer cells occupied: 2)
buffer cells:    7    5    6
               ---- ---- ----
                     W     R

Producer writes 8 (buffer cells occupied: 3)
buffer cells:    7    8    6
               ---- ---- ----
                          WR
```

Producer/Consumer Relationship **Lock & Condition Interfaces**

- **Lock and Condition interfaces**:

More precise control over thread synchronization

  - Give more precise control over thread synchronization, but are *more complicated* to use
  - Any object can contain a reference to an object that implements the **Lock** interface (of interface java.util.concurrent.locks)
  - A thread calls **Lock** method lock to acquire the lock; only one thread can hold the lock at any time, others must wait
  - A thread calls **Lock** method unlock to release the lock, allowing waiting threads to try acquire the lock
  - Class **ReentrantLock**:
    - A basic implementation of the **Lock** interface
    - fairness policy: when true, "*longest-waiting thread acquires the lock when available*"; helps avoid *indefinite postponement,* but decreases performance
  - **Condition** object:
    - Allows explicitly declaring the condition on which a thread will wait
    - Created by calling **Lock** method newCondition
    - More versatile than using *synchronized* keyword

Producer/Consumer Relationship Lock & Condition Interfaces

- **Lock and Condition interfaces**:
  - **Condition** methods await, signal, signalAll are called to wait on the **Condition** object and to let waiting threads transition to runnable state
  - **Deadlock**: occurs when a waiting thread (e.g. *thread1*) cannot proceed because it's waiting for another thread (e.g. *thread2*) to proceed, which also can't proceed because it's waiting on *thread1*
  - To prevent starvation, it's important to ensure that for every await call on a **Condition** object, there is a corresponding signal call to allow waiting threads to transition back to runnable state
  - **Lock** and **Condition** are powerful (allowing many things that can't be done using **Monitor** lock), but using them is also error-prone (e.g. not calling unlock can cause starvation for waiting threads)

Producer/Consumer Relationship Lock & Condition Interfaces

```java
// Fig. 26.20: SynchronizedBuffer.java
// Synchronizing access to a shared integer using the Lock and Condition
// interfaces
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

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

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

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

   // place int value into buffer
   public void set( int value ) throws InterruptedException
   {
      accessLock.lock(); // lock this object

      // output thread information and buffer information, then wait
      try
      {
         // while buffer is not empty, place thread in waiting state
         while ( occupied )
         {
            System.out.println( "Producer tries to write." );
            displayState( "Buffer full. Producer waits." );
            canWrite.await(); // wait until buffer is empty
         } // end while

         buffer = value; // set new buffer value

         // indicate producer cannot store another value
         // until consumer retrieves current buffer value
         occupied = true;

         displayState( "Producer writes " + buffer );

         // signal any threads waiting to read from buffer
         canRead.signalAll();
      } // end try
      finally
      {
         accessLock.unlock(); // unlock this object
      } // end finally
   } // end method set

```

Producer/Consumer Relationship Lock & Condition Interfaces

```java
   // return value from buffer
   public int get() throws InterruptedException
   {
      int readValue = 0; // initialize value read from buffer
      accessLock.lock(); // lock this object

      // output thread information and buffer information, then wait
      try
      {
         // if there is no data to read, place thread in waiting state
         while ( !occupied )
         {
            System.out.println( "Consumer tries to read." );
            displayState( "Buffer empty. Consumer waits." );
            canRead.await(); // wait until buffer is full
         } // end while

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

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

         // signal any threads waiting for buffer to be empty
         canWrite.signalAll();
      } // end try
      finally
      {
         accessLock.unlock(); // unlock this object
      } // end finally

      return readValue;
   } // end method get

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

Producer/Consumer Relationship Lock & Condition Interfaces

```java
// Fig. 26.21: SharedBufferTest2.java
// Two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

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

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

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

      // execute the Producer and Consumer tasks
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class SharedBufferTest2
```

Producer/Consumer Relationship Lock & Condition Interfaces

Operation	Buffer	Occupied
Producer writes 1	1	true
Producer tries to write. Buffer full. Producer waits.	1	true
Consumer reads 1	1	false
Producer writes 2	2	true
Producer tries to write. Buffer full. Producer waits.	2	true
Consumer reads 2	2	false
Producer writes 3	3	true
Consumer reads 3	3	false
Producer writes 4	4	true
Consumer reads 4	4	false
Consumer tries to read. Buffer empty. Consumer waits.	4	false
Producer writes 5	5	true
Consumer reads 5	5	false
Consumer tries to read. Buffer empty. Consumer waits.	5	false
Producer writes 6	6	true
Consumer reads 6	6	false
Producer writes 7	7	true
Consumer reads 7	7	false
Producer writes 8	8	true
Consumer reads 8	8	false
Producer writes 9	9	true
Consumer reads 9	9	false
Producer writes 10	10	true
Producer done producing Terminating Producer Consumer reads 10	10	false

Consumer read values totaling 55
Terminating Consumer