23

Multithreading

Do not block the way of inquiry.

— Charles Sanders Peirce

A person with one watch knows what time it is; a person with two watches is never sure.

— Proverb

Learn to labor and to wait.

— Henry Wadsworth

Longfellow

The most general definition of beauty... Multeity in Unity.

— Samuel Taylor Coleridge

The world is moving so fast these days that the man who says it can't be done is generally interrupted by someone doing it.

— Elbert Hubbard

OBJECTIVES

In this chapter you will learn:

- **What threads are and why they are useful.**
- **How threads enable you to manage concurrent activities.**
- **The life cycle of a thread.**
- **Thread priorities and scheduling.**
- **To create and execute `Runnables`.**
- **Thread synchronization.**
- **What producer/consumer relationships are and how they are implemented with multithreading.**
- **To display output from multiple threads in a Swing GUI.**
- **About `Callable` and `Future`.**

Outline

23.1 Introduction

- **Multithreading**
  - **Provides application with multiple threads of execution**
  - **Allows programs to perform tasks concurrently**
  - **Often requires programmer to synchronize threads to function correctly**

Performance Tip 23.1

A problem with single-threaded applications is that lengthy activities must complete before other activities can begin. In a multithreaded application, threads can be distributed across multiple processors (if they are available) so that multiple tasks are performed concurrently and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems that simulate concurrency—when one thread cannot proceed, another can use the processor.

Portability Tip 23.1

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

23.2 Thread States: Life Cycle of a Thread

- **Thread states**
  - ***new* state**
    - **New thread begins its life cycle in the new state**
    - **Remains in this state until program starts the thread, placing it in the *runnable* state**
  - ***runnable* state**
    - **A thread in this state is executing its task**
  - ***waiting* state**
    - **A thread transitions to this state to wait for another thread to perform a task**

23.2 Thread States: Life Cycle of a Thread

- **Thread states**
  - ***timed waiting* state**
    - **A thread enters this state to wait for another thread or for an amount of time to elapse**
    - **A thread in this state returns to the *runnable* state when it is signaled by another thread or when the timed interval expires**
  - ***terminated* state**
    - **A *runnable* thread enters this state when it completes its task**

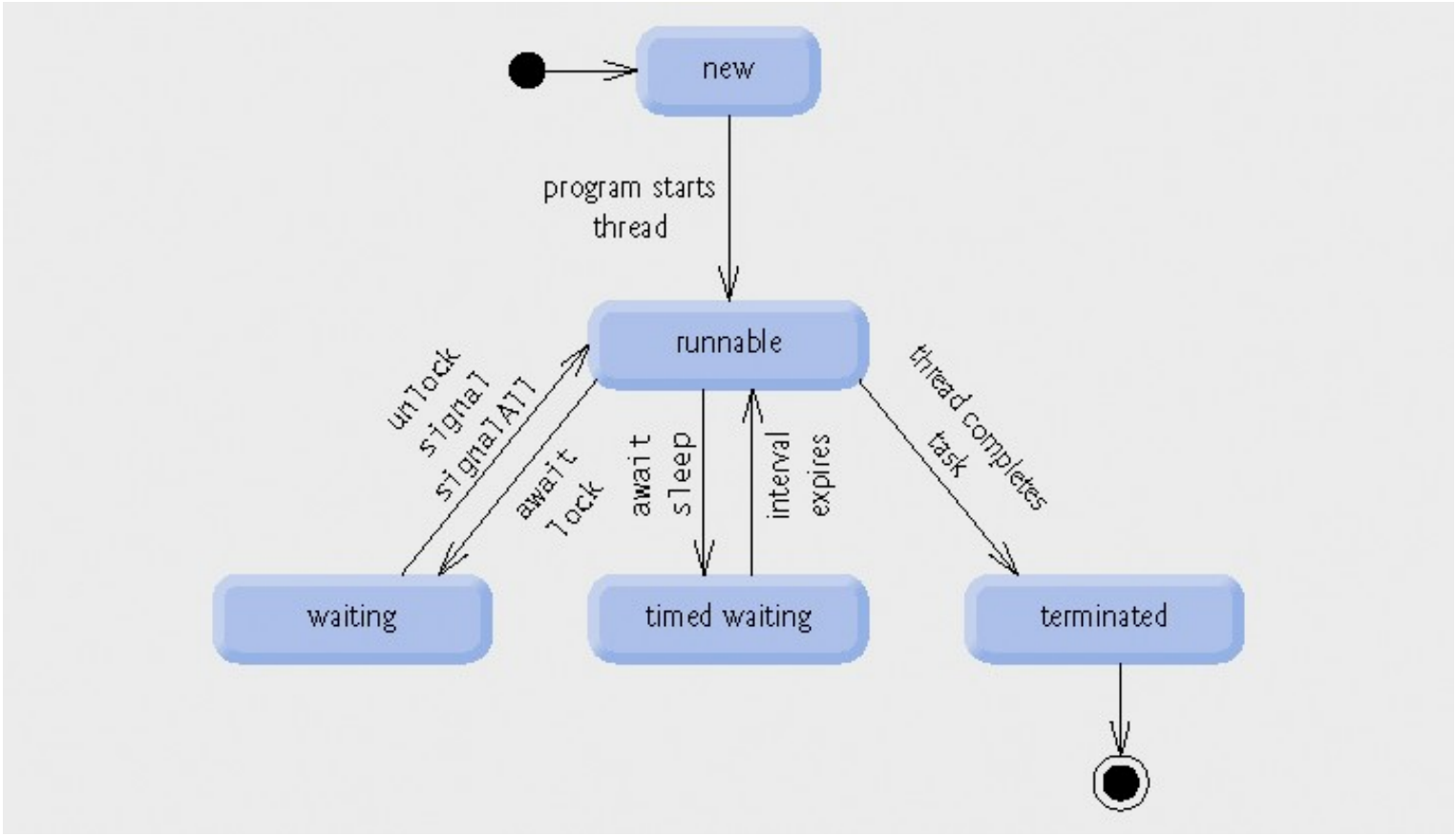


Fig. 23.1 | Thread life-cycle UML state diagram.

23.2 Thread States: Life Cycle of a Thread

- **Operating system view of *runnable* state**
  - ***ready* state**
    - **A thread in this state is not waiting for another thread, but is waiting for the operating system to assign the thread a processor**
  - ***running* state**
    - **A thread in this state currently has a processor and is executing**
  - **A thread in the *running* state often executes for a small amount of processor time called a time slice or quantum before transitioning back to the *ready* state**

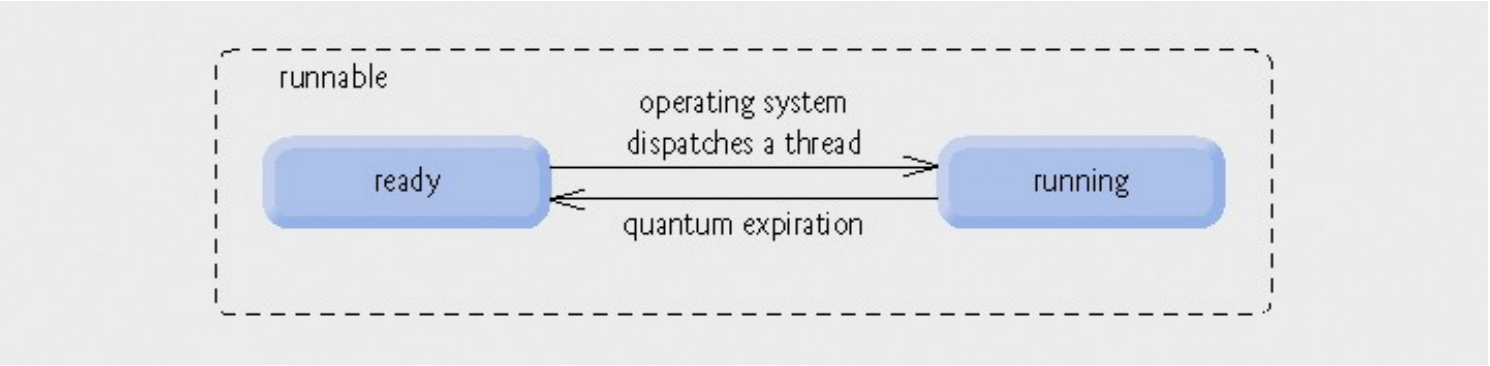


Fig. 23.2 | Operating system's internal view of Java's runnable state.

23.3 Thread Priorities and Thread Scheduling

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

23.3 Thread Priorities and Thread Scheduling

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

Portability Tip 23.2

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

Portability Tip 23.3

When designing applets and applications that use threads, you must consider the threading capabilities of all the platforms on which the applets and applications will execute.

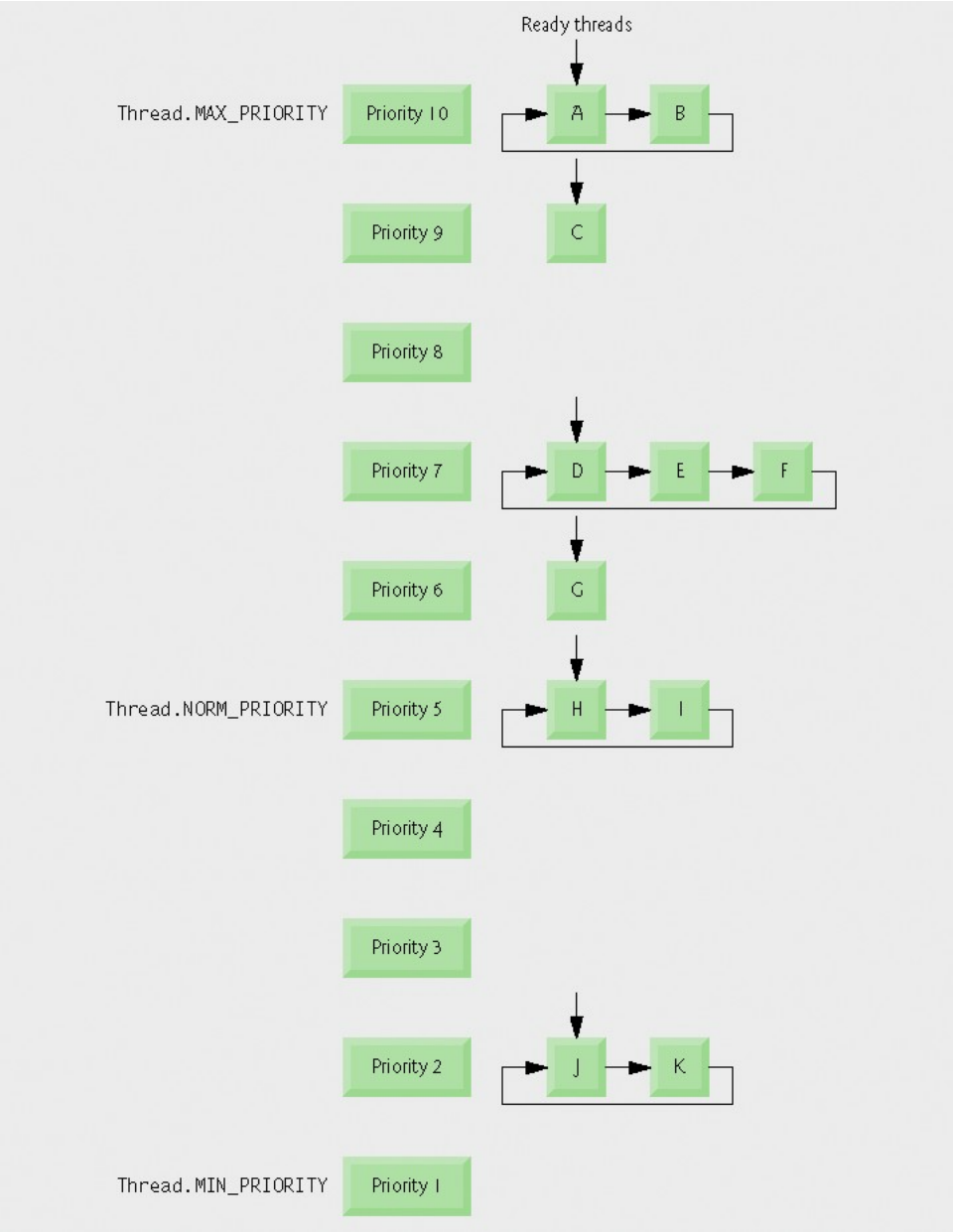


Fig. 23.3 | Thread-priority scheduling.

23.4 Creating and Executing Threads

- `Runnable` **interface**
  - **Preferred means of creating a multithreaded application**
  - **Declares method** `run`
  - **Executed by an object that implements the** `Executor` **interface**
- `Executor` **interface**
  - **Declares method** `execute`
  - **Creates and manages a group of threads called a thread pool**

23.4 Creating and Executing Threads

- `ExecutorService` **interface**
  - **Subinterface of** `Executor` **that declares other methods for managing the life cycle of an** `Executor`
  - **Can be created using** `static` **methods of class** `Executors`
  - **Method** `shutdown` **ends threads when tasks are completed**
- `Executors` **class**
  - **Method** `newFixedThreadPool` **creates a pool consisting of a fixed number of threads**
  - **Method** `newCachedThreadPool` **creates a pool that creates new threads as they are needed**

Outline

PrintTask.java

(1 of 2)

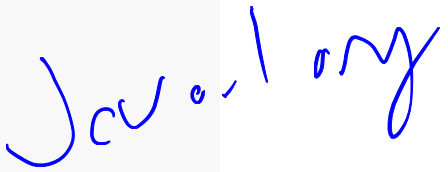
```
1  // Fig. 23.4: PrintTask.java
2  // PrintTask class sleeps for a random time from 0 to 5 seconds
3  import java.util.Random;
4
5  class PrintTask implements Runnable
6  {
7     private int sleepTime; // random sleep time for thread
8     private String threadName; // name of thread
9     private static Random generator = new Random();
10
11    // assign name to thread
12    public PrintTask( String name )
13    {
14       threadName = name; // set name of thread
15
16       // pick random sleep time between 0 and 5 seconds
17       sleepTime = generator.nextInt( 5000 );
18    } // end PrintTask constructor
19
```

Outline

PrintTask.java

(2 of 2)

```
20    // method run is the code to be executed by new thread
21    public void run()
22    {
23       try // put thread to sleep for sleepTime amount of time
24       {
25          System.out.printf( "%s going to sleep for %d milliseconds.\n",
26             threadName, sleepTime );
27
28          Thread.sleep( sleepTime ); // put thread to sleep
29       } // end try
30       // if thread interrupted while sleeping, print stack trace
31       catch ( InterruptedException exception )
32       {
33          exception.printStackTrace();
34       } // end catch
35
36       // print thread name
37       System.out.printf( "%s done sleeping\n", threadName );
38    } // end method run
39 } // end class PrintTask
```



Outline

RunnableTester .java

(1 of 2)

```
1  // Fig. 23.5: RunnableTester.java
2  // Multiple threads printing at different intervals.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5
6  public class RunnableTester
7  {
8     public static void main( String[] args )
9     {
10       // create and name each runnable
11       PrintTask task1 = new PrintTask( "thread1" );
12       PrintTask task2 = new PrintTask( "thread2" );
13       PrintTask task3 = new PrintTask( "thread3" );
14
15       System.out.println( "Starting threads" );
16
17       // create ExecutorService to manage threads
18       ExecutorService threadExecutor = Executors.newFixedThreadPool( 3 );
19
20       // start threads and place in runnable state
21       threadExecutor.execute( task1 ); // start task1
22       threadExecutor.execute( task2 ); // start task2
23       threadExecutor.execute( task3 ); // start task3
24
25       threadExecutor.shutdown(); // shutdown worker threads
26
```

```
27       System.out.println( "Threads started, main ends\n" );
28    } // end main
29 } // end class RunnableTester
```

```
Starting threads
Threads started, main ends

thread1 going to sleep for 1217 milliseconds
thread2 going to sleep for 3989 milliseconds
thread3 going to sleep for 662 milliseconds
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

```
Starting threads
thread1 going to sleep for 314 milliseconds
thread2 going to sleep for 1990 milliseconds
Threads started, main ends

thread3 going to sleep for 3016 milliseconds
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
```

Outline

RunnableTester.java

(2 of 2)

23.5 Thread Synchronization

- **Thread synchronization**
  - **Provided to the programmer with mutual exclusion**
    - **Exclusive access to a shared object**
  - **Implemented in Java using locks**
- `Lock` **interface**
  - `lock` **method obtains the lock, enforcing mutual exclusion**
  - `unlock` **method releases the lock**
  - **Class** `ReentrantLock` **implements the** `Lock` **interface**

Performance Tip 23.2

Using a `Lock` with a fairness policy helps avoid indefinite postponement, but can also dramatically reduce the overall efficiency of a program. Because of the large decrease in performance, fair locks are only necessary in extreme circumstances.

23.5 Thread Synchronization

- **Condition variables**
  - **If a thread holding the lock cannot continue with its task until a condition is satisfied, the thread can wait on a condition variable**
  - **Create by calling `Lock` method `newCondition`**
  - **Represented by an object that implements the `Condition` interface**
- **Condition interface**
  - **Declares methods `await`, to make a thread wait, `signal`, to wake up a waiting thread, and `signalAll`, to wake up all waiting threads**

Common Programming Error 23.1

Deadlock occurs when a waiting thread (let us call this thread1) cannot proceed because it is waiting (either directly or indirectly) for another thread (let us call this thread2) to proceed, while simultaneously thread2 cannot proceed because it is waiting (either directly or indirectly) for thread1 to proceed. Two threads are waiting for each other, so the actions that would enable each thread to continue execution never occur.

Error-Prevention Tip 23.1

When multiple threads manipulate a shared object using locks, ensure that if one thread calls method `await` to enter the *waiting* state for a condition variable, a separate thread eventually will call `Condition` method `signal` to transition the thread waiting on the condition variable back to the *runnable* state. (cont...)

Error-Prevention Tip 23.1

If multiple threads may be waiting on the condition variable, a separate thread can call `Condition` method `signalAll` as a safeguard to ensure that all the waiting threads have another opportunity to perform their tasks. If this is not done, indefinite postponement or deadlock could occur.

Software Engineering Observation 23.1

The locking that occurs with the execution of the `lock` and `unlock` methods could lead to deadlock if the locks are never released. Calls to method `unlock` should be placed in `finally` blocks to ensure that locks are released and avoid these kinds of deadlocks.

Performance Tip 23.3

Synchronization to achieve correctness in multithreaded programs can make programs run more slowly, as a result of thread overhead and the frequent transition of threads between the *waiting* and *runnable* states. There is not much to say, however, for highly efficient yet incorrect multithreaded programs!

Common Programming Error 23.2

It is an error if a thread issues an `await`, a `signal`, or a `signalAll` on a condition variable without having acquired the lock for that condition variable. This causes an IllegalMonitorStateException.

23.6 Producer/Consumer Relationship without Synchronization

- **Producer/consumer relationship**
  - **Producer generates data and stores it in shared memory**
  - **Consumer reads data from shared memory**
  - **Shared memory is called the buffer**

Outline

Buffer.java

```
1 // Fig. 23.6: Buffer.java
2 // Buffer interface specifies methods called by Producer and Consumer.
3
4 public interface Buffer
5 {
6    public void set( int value ); // place int value into Buffer
7    public int get(); // return int value from Buffer
8 } // end interface Buffer
```

Fig. 23.6 **`Buffer` interface used in producer/consumer examples.**

Outline

Producer.java

```
1  // Fig. 23.7: Producer.java
2  // Producer's run method stores the values 1 to 10 in buffer.
3  import java.util.Random;
4
5  public class Producer implements Runnable
6  {
7     private static Random generator = new Random();
8     private Buffer sharedLocation; // reference to shar
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
```

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

Declare `run` method to satisfy interface

Outline

Producer.java

(2 of 2)

```
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 ) ); // sleep thread
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 sleeping thread interrupted, print stack trace
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // end catch
35       } // end for
36
37       System.out.printf( "\n%s\n%s\n", "Producer done producing.",
38          "Terminating Producer." );
39    } // end method run
40 } // end class Producer
```

Sleep for up to 3 seconds

Outline

```
1  // Fig. 23.8: Consumer.java
2  // Consumer's run method loops ten times reading a value from buffer.
3  import java.util.Random;
4
5  public class Consumer implements Runnable
6  {
7     private static Random generator = new Random();
8     private Buffer sharedLocation; // reference to shar
9
10    // constructor
11    public Consumer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // end Consumer constructor
15
16    // read sharedLocation's value four times and sum t
17    public void run()
18    {
19       int sum = 0;
20
```

Consumer.java

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

Declare `run` method to satisfy interface

Outline

Consumer.java

(2 of 2)

```
21       for ( int count = 1; count <= 10; count++ )
22       {
23          // sleep 0 to 3 seconds, read value from buffer and add to sum
24          try
25          {
26             Thread.sleep( generator.nextInt( 3000 ) );
27             sum += sharedLocation.get();
28             System.out.printf( "\t\t\t%2d\n", sum );
29          } // end try
30          // if sleeping thread interrupted, print stac
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // end catch
35       } // end for
36
37       System.out.printf( "\n%s %d.\n%s\n",
38          "Consumer read values totaling", sum, "Terminating Consumer." );
39    } // end method run
40 } // end class Consumer
```

Sleep for up to 3 seconds

Outline

```
1  // Fig. 23.9: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer represents a single shared integer.
3
4  public class UnsynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7
8     // place value into buffer
9     public void set( int value )
10    {
11       System.out.printf( "Producer writes\t%2d", value );
12       buffer = value;
13    } // end method set
14
15    // return value from buffer
16    public int get()
17    {
18       System.out.printf( "Consumer reads\t%2d", buffer );
19       return buffer;
20    } // end method get
21 } // end class UnsynchronizedBuffer
```

UnsynchronizedBuffer.java

Shared variable to store data

Set the value of the buffer

Read the value of the buffer

Outline

SharedBufferTest.java

(1 of 4)

```
1  // Fig 23.10: SharedBufferTest.java
2  // Application shows two threads manipulating an unsynchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // create new thread pool with two threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // create UnsynchronizedBuffer to store ints
14       Buffer sharedLocation = new UnsynchronizedBuffer();
15
```

Create shared `UnsynchronizedBuffer` for producer and consumer to use

Outline

SharedBufferTest.java

(2 of 4)

```
16       System.out.println( "Action\t\tValue\tProduced\tConsumed" );
17       System.out.println( "------\t\t-----\t--------\t--------\n" );
18
19       // try to start producer and consumer giving each of them access
20       // to sharedLocation
21       try
22       {
23          application.execute( new Producer( sharedLocation ) );
24          application.execute( new Consumer( sharedLocation ) );
25       } // end try
26       catch ( Exception exception )
27       {
28          exception.printStackTrace();
29       } // end catch
30
31       application.shutdown(); // terminate application when threads end
32    } // end main
33 } // end class SharedBufferTest
```

Pass shared buffer to both producer and consumer

```
Action          Value   Produced        Consumed
------          -----   --------        --------

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

Producer done producing.
Terminating Producer.
Consumer reads  10                       47
Consumer reads  10                       57
Consumer reads  10                       67
Consumer reads  10                       77

Consumer read values totaling 77.
Terminating Consumer.
```

Outline

SharedBufferTest.java

(3 of 4)

```
Action          Value   Produced        Consumed
------          -----   --------        --------

Consumer reads  -1                      -1
Producer writes  1       1
Consumer reads   1                       0
Consumer reads   1                       1
Consumer reads   1                       2
Consumer reads   1                       3
Consumer reads   1                       4
Producer writes  2       3
Consumer reads   2                       6
Producer writes  3       6
Consumer reads   3                       9
Producer writes  4      10
Consumer reads   4                      13
Producer writes  5      15
Producer writes  6      21
Consumer reads   6                      19

Consumer read values totaling 19.
Terminating Consumer.
Producer writes  7      28
Producer writes  8      36
Producer writes  9      45
Producer writes 10      55

Producer done producing.
Terminating Producer.
```

Outline

SharedBufferTest.java

(4 of 4)

23.7 Producer/Consumer Relationship with Synchronization

- **Producer/consumer relationship**
  - **This example uses `Locks` and `Conditions` to implement synchronization**

Outline

```
1  // Fig. 23.11: SynchronizedBuffer.java
2  // SynchronizedBuffer synchronizes access to a single shared integer.
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5  import java.util.concurrent.locks.Condition;
6
7  public class SynchronizedBuffer implements Buffer
8  {
9     // Lock to control synchronization with this buffer
10    private Lock accessLock = new ReentrantLock();
11
12    // conditions to control reading and writing
13    private Condition canWrite = accessLock.newCondition();
14    private Condition canRead = accessLock.newCondition();
15
16    private int buffer = -1; // shared by producer and c
17    private boolean occupied = false; // whether buffer
18
19    // place int value into buffer
20    public void set( int value )
21    {
22       accessLock.lock(); // lock this object
23
```

SynchronizedBuffer

(1 of 5)

Create `ReentrantLock` for mutual exclusion

Create two `Condition` variables; one for writing and one for reading

Buffer shared by producer and consumer

Try to obtain the lock before setting the value of the shared data

Outline

SynchronizedBuffer.java

```
24      // output thread information and buffer information, then wait
25      try
26      {
27         // while buffer is not empty, place thread in waiting state
28         while ( occupied )
29         {
30            System.out.println( "Producer tries to write." );
31            displayState( "Buffer full. Producer waits." );
32            canWrite.await(); // wait until buffer is e
33         } // end while
34
35         buffer = value; // set new buffer value
36
37         // indicate producer cannot store another value
38         // until consumer retrieves current buffer value
39         occupied = true;
40
```

Producer waits until buffer is empty

Outline

SynchronizedBuffer.java

(3 of 5)

```
41          displayState( "Producer writes " + buffer );
42
43          // signal thread waiting to read from buffer
44          canRead.signal();
45       } // end try
46       catch ( InterruptedException exception )
47       {
48          exception.printStackTrace();
49       } // end catch
50       finally
51       {
52          accessLock.unlock(); // unlock this object
53       } // end finally
54    } // end method set
55
56    // return value from buffer
57    public int get()
58    {
59       int readValue = 0; // initialize value read from buffer
60       accessLock.lock(); // lock this object
61
```

Signal consumer that it may read a value

Release lock on shared data

Acquire lock before reading a value

Outline

SynchronizedBuffer.java

```
62       // output thread information and buffer information, then wait
63       try
64       {
65          // while no data to read, place thread in waiting state
66          while ( !occupied )
67          {
68             System.out.println( "Consumer tries to read." );
69             displayState( "Buffer empty. Consumer waits." );
70             canRead.await(); // wait until buffer is f
71          } // end while
72
73          // indicate that producer can store another value
74          // because consumer just retrieved buffer value
75          occupied = false;
76
77          readValue = buffer; // retrieve value from buffer
78          displayState( "Consumer reads " + readValue );
79
```

Consumer waits until buffer contains data to read

```
80          // signal thread waiting for buffer to be empty
81          canWrite.signal(); // signal producer that it can write to buffer
82       } // end try
83       // if waiting thread interrupted, print stack trace
84       catch ( InterruptedException exception )
85       {
86          exception.printStackTrace();
87       } // end catch
88       finally
89       {
90          accessLock.unlock(); // unlock this object
91       } // end finally
92
93       return readValue;
94    } // end method get
95
96    // display current operation and buffer state
97    public void displayState( String operation )
98    {
99       System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
100         occupied );
101   } // end method displayState
102} // end class SynchronizedBuffer
```

Signal producer that it can write to buffer

Release lock on shared data

SynchronizedBuffer.java

(5 of 5)

Common Programming Error 23.3

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

Software Engineering Observation 23.2

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

Common Programming Error 23.4

Forgetting to `signal` a thread that is waiting for a condition is a logic error. The thread will remain in the *waiting* state, which will prevent the thread from doing any further work. Such waiting can lead to indefinite postponement or deadlock.

Outline

SharedBufferTest2.java

(1 of 4)

```
1  // Fig 23.12: SharedBufferTest2.java
2  // Application shows two threads manipulating a synchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8     public static void main( String[] args )
9     {
10       // create new thread pool with two threads
11       ExecutorService application = Executors.newFixedThreadPool( 2 );
12
13       // create SynchronizedBuffer to store ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
```

Create `SynchronizedBuffer` to be shared between producer and consumer

Outline

SharedBufferTest2

```
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t--------" );
18
19       try // try to start producer and consumer
20       {
21          application.execute( new Producer( sharedLocation ) );
22          application.execute( new Consumer( sharedLocation ) );
23       } // end try
24       catch ( Exception exception )
25       {
26          exception.printStackTrace();
27       } // end catch
28
29       application.shutdown();
30    } // end main
31 } // end class SharedBufferTest2
```

Execute the producer and consumer in separate threads

```
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
```

Outline

SharedBufferTest2.java

(3 of 4)

```
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.
```

Outline

SharedBufferTest2.java

(4 of 4)

23.8 Producer/Consumer Relationship Circular Buffer

- **Circular buffer**
  - **Provides extra buffer space into which producer can place values and consumer can read values**

Performance Tip 23.4

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

Outline

CircularBuffer

```
1  // Fig. 23.13: CircularBuffer.java
2  // SynchronizedBuffer synchronizes access to a single shared integer.
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5  import java.util.concurrent.locks.Condition;
6
7  public class CircularBuffer implements Buffer
8  {
9     // Lock to control synchronization with this buffer
10    private Lock accessLock = new ReentrantLock();
11
12    // conditions to control reading and writing
13    private Condition canWrite = accessLock.newConditio
14    private Condition canRead = accessLock.newCondition
15
16    private int[] buffer = { -1, -1, -1 };
17
18    private int occupiedBuffers = 0; // count number of buffers used
19    private int writeIndex = 0; // index to write next value
20    private int readIndex = 0; // index to read next value
21
22    // place value into buffer
23    public void set( int value )
24    {
25       accessLock.lock(); // lock this object
26
```

Lock to impose mutual exclusion

Condition variables to control writing and reading

Circular buffer; provides three spaces for data

Obtain the lock before writing data to the circular buffer

Outline

CircularBuffer.java

```
27      // output thread information and buffer information, then wait
28      try
29      {
30         // while no empty locations, place thread in waiting state
31         while ( occupiedBuffers == buffer.length )
32         {
33            System.out.printf( "All buffers full. Producer waits.\n" );
34            canWrite.await();// await until a buffer element is free
35         } // end while
36
37         buffer[ writeIndex ] = value; // set new buffe
38
39         // update circular write index
40         writeIndex = ( writeIndex + 1 ) % buffer.lengt
41
42         occupiedBuffers++; // one more buffer element
43         displayState( "Producer writes " + buffer[ writeIndex ] );
44         canRead.signal(); // signal threads waiting to read from buffer
45      } // end try
46      catch ( InterruptedException exception )
47      {
48         exception.printStackTrace();
49      } // end catch
50      finally
51      {
52         accessLock.unlock(); // unlock this object
53      } // end finally
54   } // end method set
55
```

Wait until a buffer space is empty

Update index; this statement imposes the circularity of the buffer

Signal waiting thread it can now read data from buffer

Release the lock

Outline

```
56    // return value from buffer
57    public int get()
58    {
59       int readValue = 0; // initialize value read from buffer
60       accessLock.lock(); // lock this object
61
62       // wait until buffer has data, then read value
63       try
64       {
65          // while no data to read, place thread in wai
66          while ( occupiedBuffers == 0 )
67          {
68             System.out.printf( "All buffers empty. Consumer waits.\n" );
69             canRead.await(); // await until a buffer element is filled
70          } // end while
71
72          readValue = buffer[ readIndex ]; // read valu
73
74          // update circular read index
75          readIndex = ( readIndex + 1 ) % buffer.length;
76
```

Buffer

Lock the object before attempting to read a value

Wait for a value to be written to the buffer

Update read index; this statement imposes the circularity of the buffer

Outline

```
77          occupiedBuffers--; // one more buffer element is empty
78          displayState( "Consumer reads " + readValue );
79          canWrite.signal(); // signal threads waiting to write to buffer
80       } // end try
81       // if waiting thread interrupted, print stack tr
82       catch ( InterruptedException exception )
83       {
84          exception.printStackTrace();
85       } // end catch
86       finally
87       {
88          accessLock.unlock(); // unlock this object
89       } // end finally
90
91       return readValue;
92    } // end method get
93
94    // display current operation and buffer state
95    public void displayState( String operation )
96    {
97       // output operation and number of occupied buffers
98       System.out.printf( "%s%s%d)\n%s", operation,
99          " (buffers occupied: ", occupiedBuffers, "buffers:  " );
100
101      for ( int value : buffer )
102         System.out.printf( " %2d  ", value ); // output values in buffer
103
```

Signal thread waiting to write to the buffer

CircularBuffer.java

Release the lock

```
104      System.out.print( "\n         " );
105      for ( int i = 0; i < buffer.length; i++ )
106         System.out.print( "---- " );
107
108      System.out.print( "\n         " );
109      for ( int i = 0; i < buffer.length; i++ )
110      {
111         if ( i == writeIndex && i == readIndex )
112            System.out.print( " WR" ); // both write and read index
113         else if ( i == writeIndex )
114            System.out.print( " W   " ); // just write index
115         else if ( i == readIndex )
116            System.out.print( "  R  " ); // just read index
117         else
118            System.out.print( "     " ); // neither index
119      } // end for
120
121      System.out.println( "\n" );
122   } // end method displayState
123} // end class CircularBuffer
```

Outline

CircularBuffer.java

(5 of 5)

Outline

CircularBufferTest.java

(1 of 4)

```
1  // Fig 23.14: CircularBufferTest.java
2  // Application shows two threads manipulating a circular buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class CircularBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // create new thread pool with two threads
11       ExecutorService application = Executors.newFixed
12
13       // create CircularBuffer to store ints
14       Buffer sharedLocation = new CircularBuffer();
15
16       try // try to start producer and consumer
17       {
18          application.execute( new Producer( sharedLocation ) );
19          application.execute( new Consumer( sharedLocation ) );
20       } // end try
21       catch ( Exception exception )
22       {
23          exception.printStackTrace();
24       } // end catch
25
26       application.shutdown();
27    } // end main
28 } // end class CircularBufferTest
```

Create `CircularBuffer` for use in both the producer and consumer

Execute the producer and consumer in separate threads

```
Producer writes 1 (buffers occupied: 1)
buffers:    1   -1   -1
         ---- ---- ----
           R   W

Consumer reads 1 (buffers occupied: 0)
buffers:    1   -1   -1
         ---- ---- ----
              WR

All buffers empty. Consumer waits.
Producer writes 2 (buffers occupied: 1)
buffers:    1    2   -1
         ---- ---- ----
                R   W

Consumer reads 2 (buffers occupied: 0)
buffers:    1    2   -1
         ---- ---- ----
                   WR

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

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

Producer writes 4 (buffers occupied: 1)
buffers:    4    2    3
         ---- ---- ----
           R   W
```

Outline

CircularBufferTest.java

(2 of 4)

```
Producer writes 5 (buffers occupied: 2)
buffers:    4    5    3
         ---- ---- ----
           R        W

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

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

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

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

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

Outline

CircularBufferTest.java

(3 of 4)

```
Consumer reads 6 (buffers occupied: 2)
buffers:    7    8    6
          ---- ---- ----
            R        W

Consumer reads 7 (buffers occupied: 1)
buffers:    7    8    6
          ---- ---- ----
                 R   W

Producer writes 9 (buffers occupied: 2)
buffers:    7    8    9
          ---- ---- ----
           W     R

Consumer reads 8 (buffers occupied: 1)
buffers:    7    8    9
          ---- ---- ----
           W          R

Consumer reads 9 (buffers occupied: 0)
buffers:    7    8    9
          ---- ---- ----
           WR

Producer writes 10 (buffers occupied: 1)
buffers:   10    8    9
          ---- ---- ----
            R   W


Producer done producing.
Terminating Producer.
Consumer reads 10 (buffers occupied: 0)
buffers:   10    8    9
          ---- ---- ----
                WR


Consumer read values totaling: 55.
Terminating Consumer.
```

Outline

CircularBufferTest.java

(4 of 4)

23.9 Producer/Consumer Relationship `ArrayBlockingQueue`

- `ArrayBlockingQueue`
  - **Fully implemented version of the circular buffer**
  - **Implements the `BlockingQueue` interface**
  - **Declares methods `put` and `take` to write and read date from the buffer, respectively**

Outline

BlockingBuffer.java

```
1  // Fig. 23.15: BlockingBuffer.java
2  // Class synchronizes access to a blocking buffer.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer
6  {
7     private ArrayBlockingQueue<Integer> buffer;
8
9     public BlockingBuffer()
10    {
11       buffer = new ArrayBlockingQueue<Integer>( 3 );
12    } // end BlockingBuffer constructor
13
14    // place value into buffer
15    public void set( int value )
16    {
17       try
18       {
19          buffer.put( value ); // place value in circular buffer
20          System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
21             "Buffers occupied: ", buffer.size() );
22       } // end try
23       catch ( Exception exception )
24       {
25          exception.printStackTrace();
26       } // end catch
27    } // end method set
28
```

Create instance of `ArrayBlockingQueue` to store data

Place a value into the buffer; blocks if buffer is full

Outline

BlockingBuffer.java

(2 of 2)

```
29    // return value from buffer
30    public int get()
31    {
32       int readValue = 0; // initialize value read from buffer
33
34       try
35       {
36          readValue = buffer.take(); // remove value from circular buffer
37          System.out.printf( "%s %2d\t%s%d\n", "Consumer reads ",
38             readValue, "Buffers occupied: ", buffer.siz
39       } // end try
40       catch ( Exception exception )
41       {
42          exception.printStackTrace();
43       } // end catch
44
45       return readValue;
46    } // end method get
47 } // end class BlockingBuffer
```

Remove value from buffer; blocks if buffer is empty

Outline

BlockingBufferTest.java

(1 of 2)

```
1  // Fig 23.16: BlockingBufferTest.java
2  // Application shows two threads manipulating a blocking buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class BlockingBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // create new thread pool with two threads
11       ExecutorService application = Executors.newFixed
12
13       // create BlockingBuffer to store ints
14       Buffer sharedLocation = new BlockingBuffer();
15
16       try // try to start producer and consumer
17       {
18          application.execute( new Producer( sharedLocation ) );
19          application.execute( new Consumer( sharedLocation ) );
20       } // end try
21       catch ( Exception exception )
22       {
23          exception.printStackTrace();
24       } // end catch
25
```

Create `BlockingBuffer` for use in producer and consumer

Execute the producer and consumer in separate threads

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

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

Producer done producing.
Terminating Producer.
Consumer reads   9      Buffers occupied: 1
Consumer reads  10      Buffers occupied: 0

Consumer read values totaling 55.
Terminating Consumer.
```

Outline

BlockingBufferTest.java

(2 of 2)

23.10 Multithreading with GUI

- **Swing GUI components**
  - **Not thread safe**
  - **Updates should be performed in the event-dispatching thread**
    - **Use `static` method `invokeLater` of class `SwingUtilities` and pass it a `Runnable` object**

Outline

```
1  // Fig. 23.17: RunnableObject.java
2  // Runnable that writes a random character to a JLabel
3  import java.util.Random;
4  import java.util.concurrent.locks.Condition;
5  import java.util.concurrent.locks.Lock;
6  import javax.swing.JLabel;
7  import javax.swing.SwingUtilities;
8  import java.awt.Color;
9
10 public class RunnableObject implements Runnable
11 {
12    private static Random generator = new Random(); // for random letters
13    private Lock lockObject; // application lock; passe
14    private Condition suspend; // used to suspend and r
15    private boolean suspended = false; // true if threa
16    private JLabel output; // JLabel for output
17
18    public RunnableObject( Lock theLock, JLabel label )
19    {
20       lockObject = theLock; // store the Lock for the application
21       suspend = lockObject.newCondition(); // create new Condition
22       output = label; // store JLabel for outputting character
23    } // end RunnableObject constructor
24
25    // place random characters in GUI
26    public void run()
27    {
28       // get name of executing thread
29       final String threadName = Thread.currentThread().getName();
30
```

RunnableObject

Implement the `Runnable` interface

`Lock` to implement mutual exclusion

`Condition` variable for suspending the threads

Boolean to control whether thread is suspended

Create the `Lock` and a `Condition` variable

Get name of current thread

Outline

```
31    while ( true ) // infinite loop; will be terminated from outside
32    {
33       try
34       {
35          // sleep for up to 1 second
36          Thread.sleep( generator.nextInt( 1000 ) );
37
38          lockObject.lock(); // obtain the lock
39          try
40          {
41             while ( suspended ) // loop until not s
42             {
43                suspend.await(); // suspend thread execution
44             } // end while
45          } // end try
46          finally
47          {
48             lockObject.unlock(); // unlock the lock
49          } // end finally
50       } // end try
51       // if thread interrupted during wait/sleep
52       catch ( InterruptedException exception )
53       {
54          exception.printStackTrace(); // print stack trace
55       } // end catch
56
```

Obtain the lock to impose mutual exclusion

Wait while thread is suspended

Release the lock

Object .java

(2 of 4)

```
57          // display character on corresponding JLabel
58          SwingUtilities.invokeLater(
59             new Runnable()
60             {
61                // pick random character and display it
62                public void run()
63                {
64                   // select random uppercase letter
65                   char displayChar =
66                      ( char ) ( generator.nextInt( 26 ) + 65 );
67
68                   // output character in JLabel
69                   output.setText( threadName + ": " + displayChar );
70                } // end method run
71             } // end inner class
72          ); // end call to SwingUtilities.invokeLater
73       } // end while
74    } // end method run
75
```

Call `invokeLater`

Method `invokeLater` is passed a `Runnable`

Object

.java

(3 of 4)

Outline

RunnableObject .java

(4 of 4)

```
76    // change the suspended/running state
77    public void toggle()
78    {
79       suspended = !suspended; // toggle boolean controlling state
80
81       // change label color on suspend/resume
82       output.setBackground( suspended ? Color.RED : Color.GREEN );
83
84       lockObject.lock(); // obtain lock
85       try
86       {
87          if ( !suspended ) // if thread resumed
88          {
89             suspend.signal(); // resume thread
90          } // end if
91       } // end try
92       finally
93       {
94          lockObject.unlock(); // release lock
95       } // end finally
96    } // end method toggle
97 } // end class RunnableObject
```

Obtain lock for the application

Resume a waiting thread

Release the lock

Outline

RandomCharacters.java

(1 of 4)

```
1  // Fig. 23.18: RandomCharacters.java
2  // Class RandomCharacters demonstrates the Runnable interface
3  import java.awt.Color;
4  import java.awt.GridLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import java.util.concurrent.Executors;
8  import java.util.concurrent.ExecutorService;
9  import java.util.concurrent.locks.Condition;
10 import java.util.concurrent.locks.Lock;
11 import java.util.concurrent.locks.ReentrantLock;
12 import javax.swing.JCheckBox;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15
16 public class RandomCharacters extends JFrame implements ActionListener
17 {
18    private final static int SIZE = 3; // number of threads
19    private JCheckBox checkboxes[]; // array of JCheckBoxes
20    private Lock lockObject = new ReentrantLock( true ); // single lock
21
22    // array of RunnableObjects to display random characters
23    private RunnableObject[] randomCharacters =
24       new RunnableObject[ SIZE ];
25
```

Create `Lock` for the application

Outline

RandomCharacters.java

```
26    // set up GUI and arrays
27    public RandomCharacters()
28    {
29       checkboxes = new JCheckBox[ SIZE ]; // allocate space for array
30       setLayout( new GridLayout( SIZE, 2, 5, 5 ) ); // set layout
31
32       // create new thread pool with SIZE threads
33       ExecutorService runner = Executors.newFixedThreadPool( SIZE );
34
35       // loop SIZE times
36       for ( int count = 0; count < SIZE; count++ )
37       {
38          JLabel outputJLabel = new JLabel(); // create JLabel
39          outputJLabel.setBackground( Color.GREEN ); // set color
40          outputJLabel.setOpaque( true ); // set JLabel to be opaque
41          add( outputJLabel ); // add JLabel to JFrame
42
43          // create JCheckBox to control suspend/resume state
44          checkboxes[ count ] = new JCheckBox( "Suspended" );
45
46          // add listener which executes when JCheckBox is clicked
47          checkboxes[ count ].addActionListener( this );
48          add( checkboxes[ count ] ); // add JCheckBox to JFrame
49
```

Create thread pool for executing threads

Outline

RandomCharacters

(3 of 4)

```
50          // create a new RunnableObject
51          randomCharacters[ count ] =
52             new RunnableObject( lockObject, outputJLabel );
53
54          // execute RunnableObject
55          runner.execute( randomCharacters[ count ] );
56       } // end for
57
58       setSize( 275, 90 ); // set size of window
59       setVisible( true ); // show window
60
61       runner.shutdown(); // shutdown runner when threads finish
62    } // end RandomCharacters constructor
63
64    // handle JCheckBox events
65    public void actionPerformed( ActionEvent event )
66    {
67       // loop over all JCheckBoxes in array
68       for ( int count = 0; count < checkboxes.length; count++ )
69       {
70          // check if this JCheckBox was source of event
71          if ( event.getSource() == checkboxes[ count ] )
72             randomCharacters[ count ].toggle(); // toggle state
73       } // end for
74    } // end method actionPerformed
75
```

Execute a `Runnable`

Shutdown thread pool when threads finish their tasks

Outline

RandomCharacters.java

(4 of 4)

```
76    public static void main( String args[] )
77    {
78       // create new RandomCharacters object
79       RandomCharacters application = new RandomCharacters();
80
81       // set application to end when window is closed
82       application.setDefaultCloseOperation( EXIT_ON_CLOSE );
83    } // end main
84 } // end class RandomCharacters
```

23.11 Other Classes and Interfaces in `java.util.concurrent`

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

23.12 Monitors and Monitor Locks

- **Monitors**
  - **Every object in Java has a monitor**
  - **Allows one thread at a time to execute inside a `synchronized` statement**
  - **Threads waiting to acquire the monitor lock are placed in the *blocked* state**
  - **`Object` method `wait` places a thread in the waiting state**
  - **`Object` method `notify` wakes up a waiting thread**
  - **`Object` method `notifyAll` wakes up all waiting threads**

Software Engineering Observation 23.3

The locking that occurs with the execution of `synchronized` methods could lead to deadlock if the locks are never released. When exceptions occur, Java's exception mechanism coordinates with Java's synchronization mechanism to release locks and avoid these kinds of deadlocks.

Common Programming Error 23.5

It is an error if a thread issues a `wait`, a `notify` or a `notifyAll` on an object without having acquired a lock for it. This causes an `IllegalMonitorStateException`.

Outline

SynchronizedBuffer.java

(1 of 3)

```
1  // Fig. 23.19: SynchronizedBuffer.java
2  // SynchronizedBuffer synchronizes access to a single shared integer.
3
4  public class SynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7     private boolean occupied = false; // count of occupied buffers
8
9     // place value into buffer
10    public synchronized void set( int value )
11    {
12       // while there are no empty locations, place thr
13       while ( occupied )
14       {
15          // output thread information and buffer information, then wait
16          try
17          {
18             System.out.println( "Producer tries to write." );
19             displayState( "Buffer full. Producer waits." );
20             wait();
21          } // end try
22          catch ( InterruptedException exception )
23          {
24             exception.printStackTrace();
25          } // end catch
26       } // end while
27
28       buffer = value; // set new buffer value
29
```

Declare a `synchronized` set method

Wait until buffer is empty

Set buffer value

Outline

```
30       // indicate producer cannot store another value
31       // until consumer retrieves current buffer value
32       occupied = true;
33
34       displayState( "Producer writes " + buffer );
35
36       notify(); // tell waiting thread to enter runnable state
37    } // end method set; releases lock on SynchronizedBuffer
38
39    // return value from buffer
40    public synchronized int get()
41    {
42       // while no data to read, place thread in waitin
43       while ( !occupied )
44       {
45          // output thread information and buffer information, then wait
46          try
47          {
48             System.out.println( "Consumer tries to read." );
49             displayState( "Buffer empty. Consumer waits." );
50             wait();
51          } // end try
52          catch ( InterruptedException exception )
53          {
54             exception.printStackTrace();
55          } // end catch
56       } // end while
57
```

Buffer is now occupied

SynchronizedBuffer.java

Notify waiting thread that it may now read a value

Declare `synchronized` get method

Wait until buffer is full

Outline

```
58       // indicate that producer can store another value
59       // because consumer just retrieved buffer value
60       occupied = false;
61
62       int readValue = buffer; // store value in buffer
63       displayState( "Consumer reads " + readValue );
64
65       notify(); // tell waiting thread to enter runnable state
66
67       return readValue;
68    } // end method get; releases lock on SynchronizedBu
69
70    // display current operation and buffer state
71    public void displayState( String operation )
72    {
73       System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
74          occupied );
75    } // end method displayState
76 } // end class SynchronizedBuffer
```

Buffer is now empty

Notify thread it may now write to buffer

SynchronizedBuffer.java

Outline

SharedBufferTest2.java

(1 of 3)

```
1  // Fig 23.20: SharedBufferTest2.java
2  // Application shows two threads manipulating a synchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8     public static void main( String[] args )
9     {
10       // create new thread pool with two threads
11       ExecutorService application = Executors.newFixed
12
13       // create SynchronizedBuffer to store ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t
18
19       try // try to start producer and consumer
20       {
21          application.execute( new Producer( sharedLocation ) );
22          application.execute( new Consumer( sharedLocation ) );
23       } // end try
24       catch ( Exception exception )
25       {
26          exception.printStackTrace();
27       } // end catch
28
```

Create `SynchronizedBuffer` for use in producer and consumer

Execute the producer and consumer in separate threads

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

```
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

Consumer tries to read.
Buffer empty. Consumer waits.           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
```

Outline

SynchronizedBuffer.java

(2 of 3)

```
Producer writes 6                   6               true

Consumer reads 6                    6               false

Consumer tries to read.
Buffer empty. Consumer waits.       6               false

Producer writes 7                   7               true

Consumer reads 7                    7               false

Consumer tries to read.
Buffer empty. Consumer waits.       7               false

Producer writes 8                   8               true

Consumer reads 8                    8               false

Producer writes 9                   9               true

Producer tries to write.
Buffer full. Producer waits.        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.
```

Outline

SynchronizedBuffer.java

(3 of 3)