23

Multithreading

The most general definition of beauty …Multeity in Unity.

—Samuel Taylor Coleridge

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 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 `Runnable`s.
- Thread synchronization.
- What producer/consumer relationships are and how they are implemented with multithreading.
- To enable multiple threads to update Swing GUI components in a thread-safe manner.
- About interfaces `Callable` and `Future`, which you can use with threading to execute tasks that return results.

Outline

23.1 Introduction

23.2 Thread States: Life Cycle of a Thread

23.3 Thread Priorities and Thread Scheduling

23.4 Creating and Executing Threads

23.4.1 `Runnables` and the `Thread` Class

23.4.2 Thread Management with the `Executor` Framework

23.5 Thread Synchronization

23.6 Producer/Consumer Relationship without Synchronization

23.5.1 Unsynchronized Data Sharing

23.5.2 Synchronized Data Sharing—Making Operations Atomic

Outline

23.1 Introduction

- **The human body performs a great variety of operations in parallel—or concurrently**
- **Computers, too, can perform operations concurrently**
- **Only computers that have multiple processors can truly execute multiple instructions concurrently**
- **Operating systems on single-processor computers create the illusion of concurrent execution by rapidly switching between activities, but on such computers only a single instruction can execute at once**

23.1 Introduction

- Most programming languages do not enable you to specify concurrent activities
- Historically, concurrency has been implemented with operating system primitives available only to experienced systems programmers
- Ada made concurrency primitives widely available to defense contractors building military command-and-control systems
  - not widely used in academia and industry
- Java makes concurrency available to you through the language and APIs

Performance Tip 23.1

A problem with single-threaded applications that can lead to poor responsiveness is that lengthy activities must complete before others can begin. In a multithreaded application, threads can be distributed across multiple processors (if available) so that multiple tasks execute 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 (because, for example, it is waiting for the result of an I/O operation), another can use the processor.

23.1 Introduction

- **An application of concurrent programming**
  - **Start playback of an audio clip or a video clip while the clip downloads**
  - **synchronize (coordinate the actions of) the threads so that the player thread doesn't begin until there is a sufficient amount of the clip in memory to keep the player thread busy**
- **The Java Virtual Machine (JVM) creates threads to run a program, the JVM also may create threads for performing housekeeping tasks such as garbage collection**
- **Programming concurrent applications is difficult and error-prone**
  - **Follow some simple guidelines**
    - ***Use existing classes from the Java API* such as the `ArrayBlockingQueue` class *that manage synchronization for you.* The classes in the Java API are written by experts, have been fully tested and debugged, operate efficiently and help you avoid common traps and pitfalls.**
    - **If you find that you need more custom functionality than that provided in the Java APIs, you should use the `synchronized` keyword and `Object` methods `wait`, `notify` and `notifyAll`**
    - **If you need even more complex capabilities, then you should use the `Lock` and `Condition` interfaces**

23.1 Introduction

- The `Lock` and `Condition` interfaces should be used only by advanced programmers who are familiar with the common traps and pitfalls of concurrent programming

23.2 Thread States: Life Cycle of a Thread

- **A thread occupies one of several thread states (Fig. 23.1)**
- **A new thread begins its life cycle in the *new* state.**
- **When the program starts the thread it enters the *runnable* state.**
  - **considered to be executing its task**
- ***Runnable* thread transitions to the *waiting* state while it waits for another thread to perform a task**
  - **transitions back to the *runnable* state only when another thread notifies the waiting thread to continue executing**
- **A *runnable* thread can enter the *timed waiting* state for a specified interval of time**
  - **transitions back to the *runnable* state when that time interval expires or when the event it is waiting for occurs.**

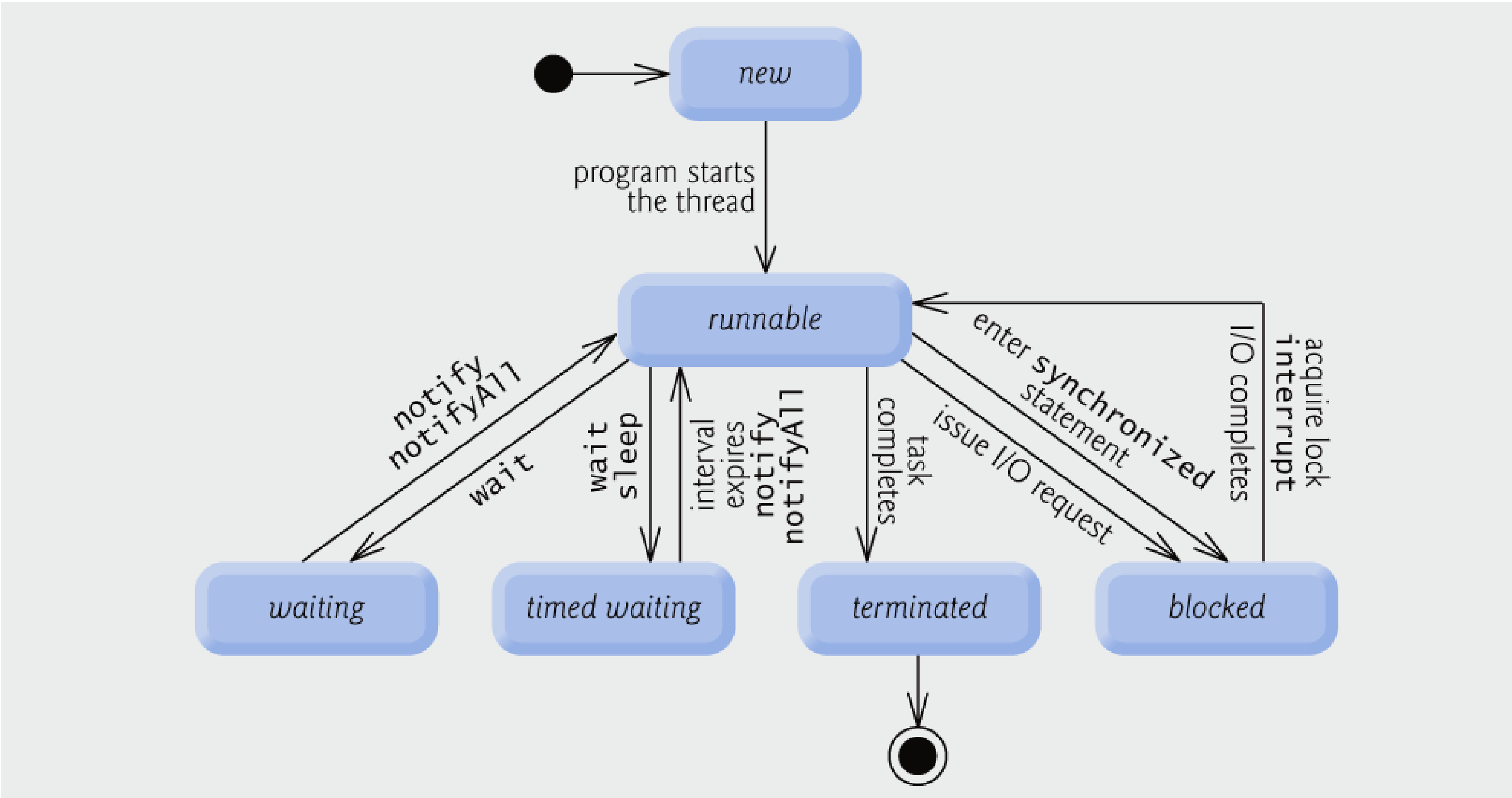


Fig. 23.1 | Thread life-cycle UML state diagram.

23.2 Thread States: Life Cycle of a Thread

- ***Timed waiting* and *waiting* threads cannot use a processor, even if one is available.**
- **A *runnable* thread can transition to the *timed waiting* state if it provides an optional wait interval when it is waiting for another thread to perform a task.**
  - **returns to the *runnable* state when**
    - **it is notified by another thread, or**
    - **the timed interval expires**
- **A thread also enters the *timed waiting* state when put to sleep**
  - **remains in the *timed waiting* state for a designated period of time then returns to the *runnable* state**
- **A *runnable* thread transitions to the *blocked* state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes.**
  - **A *blocked* thread cannot use a processor, even if one is available**
- **A *runnable* thread enters the *terminated* state (sometimes called the *dead* state) when it successfully completes its task or otherwise terminates (perhaps due to an error).**

23.2 Thread States: Life Cycle of a Thread

- **At the operating system level, Java's runnable state typically encompasses two separate states (Fig. 23.2).**
  - **Operating system hides these states from the JVM**
  - **A *runnable* thread first enters the *ready* state**
  - **When thread is dispatched by the OS it enters the *running* state**
  - **When the thread's quantum expires, the thread returns to the *ready* state and the operating system dispatches another thread**
  - **Transitions between the ready and running states are handled solely by the operating system**

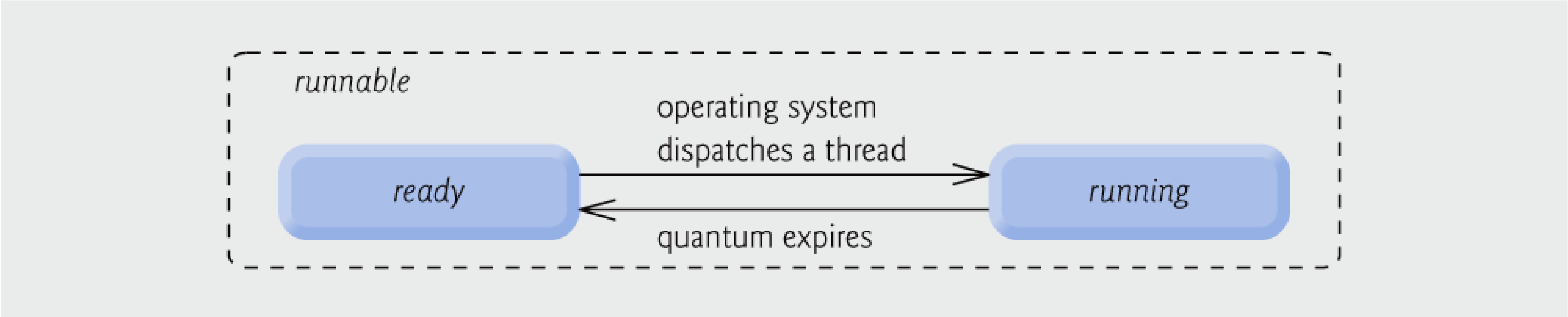


Fig. 23.2 | Operating system's internal view of Java's *runnable* state.

23.3 Thread Priorities and Thread Scheduling

- **Every Java thread has a thread priority that helps the operating system determine the order in which threads are scheduled**
- **Priorities range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10)**
- **By default, every thread is given priority NORM_PRIORITY (a constant of 5)**
- **Each new thread inherits the priority of the thread that created it**

23.3 Thread Priorities and Thread Scheduling

- Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads
  - Does not guarantee the order in which threads execute
- Timeslicing
  - enables threads of equal priority to share a processor
  - when thread's quantum expires, processor is given to the next thread of equal priority, if one is available
- Thread scheduler determines which thread runs next
- Higher-priority threads generally preempt the currently running threads of lower priority
  - known as preemptive scheduling
  - Possible indefinite postponement (starvation)

Portability Tip 23.1

Thread scheduling is platform dependent—the behavior of a multithreaded program could vary across different Java implementations.

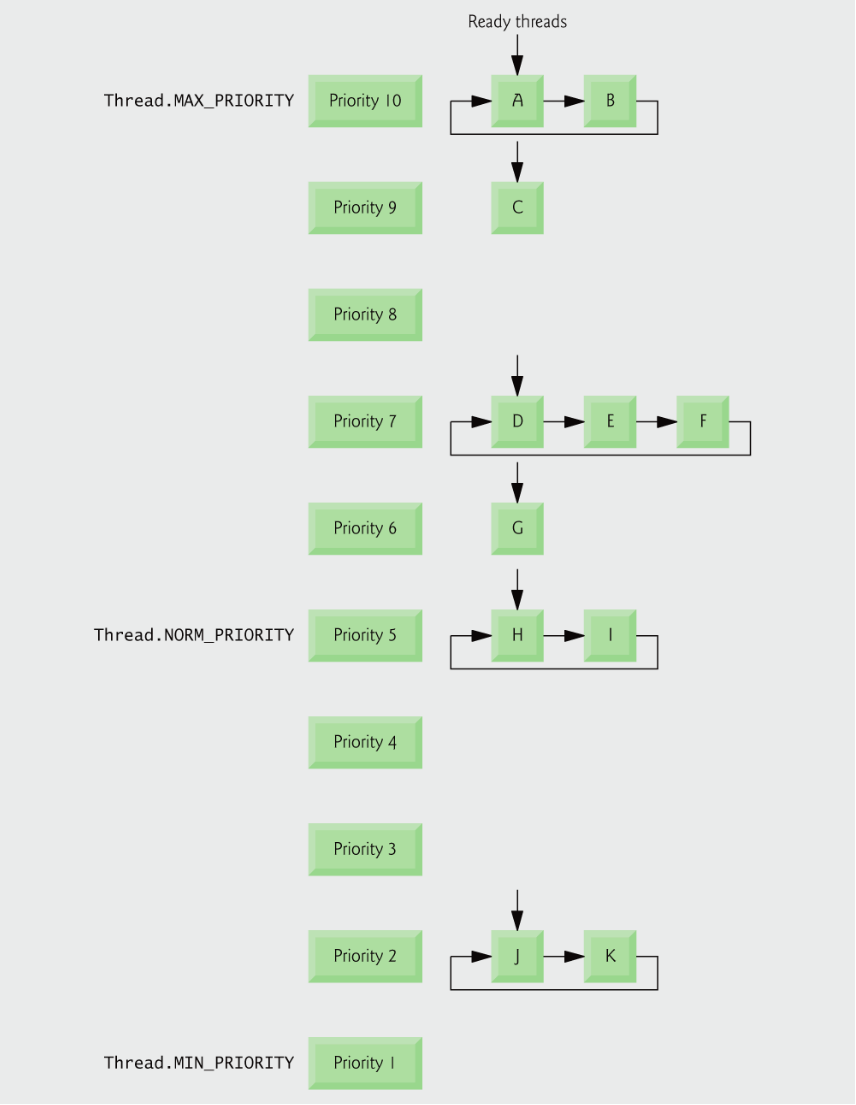


Fig. 23.3 | Thread-priority scheduling.

Portability Tip 23.2

When designing multithreaded programs consider the threading capabilities of all the platforms on which the programs will execute. Using priorities other than the default will make your programs' behavior platform specific. If portability is your goal, don't adjust thread priorities.

23.4 Creating and Executing Threads

- `Runnable` interface (of package `java.lang`)
- `Runnable` object represents a “task” that can execute concurrently with other tasks
- Method `run` contains the code that defines the task that a `Runnable` object should perform

23.4.1 Runnables and the Thread Class

- **Method `sleep` throws a (checked) `InterruptedException` if the sleeping thread's `interrupt` method is called**
- **The code in method `main` executes in the main thread, a thread created by the JVM**

```java
// Fig. 23.4: 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();

   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

```

Implement `Runnable` to define a task that can execute concurrently

```java
   // method run contains the co
   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
```

Define task in method `run`

Outline

```
// Fig. 23.5: ThreadCreator.java
// Creating and starting three threads to execute Runnables.
import java.lang.Thread;

public class ThreadCreator
{
   public static void main( String[] args )
   {
      System.out.println( "Creating threads" );

      // create each thread with a new targeted runnable
      Thread thread1 = new Thread( new PrintTask( "task1" ) );
      Thread thread2 = new Thread( new PrintTask( "task2" ) );
      Thread thread3 = new Thread( new PrintTask( "task3" ) );

      System.out.println( "Threads created, starting tasks." );

      // start threads and place in runnable state
      thread1.start(); // invokes task1's run method
      thread2.start(); // invokes task2's run method
      thread3.start(); // invokes task3's run method

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

ThreadCreator .java

(1 of 2)

Create `Thread`s to execute each new `Runnable` object

Start the `Thread`s to begin processing the concurrent tasks

```
Creating threads
Threads created, starting tasks
Tasks started, main ends

task3 going to sleep for 491 milliseconds
task2 going to sleep for 71 milliseconds
task1 going to sleep for 3549 milliseconds
task2 done sleeping
task3 done sleeping
task1 done sleeping
```

```
Creating threads
Threads created, starting tasks
task1 going to sleep for 4666 milliseconds
task2 going to sleep for 48 milliseconds
task3 going to sleep for 3924 milliseconds
Tasks started, main ends

thread2 done sleeping
thread3 done sleeping
thread1 done sleeping
```

ThreadCreator .java

(2 of 2)

23.4.2 Thread Management with the Executor Framework

- Recommended that you use the `Executor` interface to manage the execution of `Runnable` objects
- An `Executor` object creates and manages a thread pool to execute `Runnable`s
- `Executor` advantages over creating threads yourself
  - Reuse existing threads to eliminate new thread overhead
  - Improve performance by optimizing the number of threads to ensure that the processor stays busy
- `Executor` method `execute` accepts a `Runnable` as an argument
  - Assigns each `Runnable` it receives to one of the available threads in the thread pool
  - If none available, creates a new thread or waits for a thread to become available

23.4.2 Thread Management with the Executor Framework

- Interface `ExecutorService`
  - package `java.util.concurrent`
  - extends `Executor`
  - declares methods for managing the life cycle of an `Executor`
  - Objects of this type are created using `static` methods declared in class `Executors` (of package `java.util.concurrent`)
- `Executors` method `newCachedThreadPool` obtains an `ExecutorService` that creates new threads as they are needed
- `ExecutorService` method `execute` returns immediately from each invocation
- `ExecutorService` method `shutdown` notifies the `ExecutorService` to stop accepting new tasks, but continues executing tasks that have already been submitted

```java
// Fig. 23.6: 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();

```


Creates an `ExecutorService` that manages a cached thread pool

```java
      // 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 th
      threadExecutor.shutdown();

      System.out.println( "Tasks started,
   } // end main
} // end class TaskExecutor
```

Use the `ExecutorService`'s `execute` method to assign each new `Runnable` object to a thread thread pool

Prevent the `ExecutorService` from accepting additional `Runnable` objects

`.java`

(2 of 2)

```
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
thread3 done sleeping
thread2 done sleeping
thread1 done sleeping
```

```
Starting Executor
task1 going to sleep for 1342 milliseconds
task2 going to sleep for 277 milliseconds
task3 going to sleep for 2737 milliseconds
Tasks started, main ends

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

23.5 Thread Synchronization

- **Coordinates access to shared data by multiple concurrent threads**
  - **Indeterminate results may occur unless access to a shared object is managed properly**
  - **Give only one thread at a time exclusive access to code that manipulates a shared object**
  - **Other threads wait**
  - **When the thread with exclusive access to the object finishes manipulating the object, one of the threads that was waiting is allowed to proceed**
- **Mutual exclusion**

23.5 Thread Synchronization

- **Java provides built-in monitors to implement synchronization**
- **Every object has a monitor and a monitor lock.**
  - **Monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time**
  - **Can be used to enforce mutual exclusion**
- **To enforce mutual exclusion**
  - **thread must acquire the lock before it can proceed with its operation**
  - **other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock**

23.5 Thread Synchronization

- `synchronized` statement
  - Enforces mutual exclusion on a block of code
  - ```
    synchronized ( object )
    {
        statements
    } // end synchronized statement
    ```
  - where *`object`* is the object whose monitor lock will be acquired (normally `this`)
- A `synchronized` method is equivalent to a `synchronized` statement that encloses the entire body of a method
    ```

23.5.1 Unsynchronized Data Sharing

- `ExecutorService` method `awaitTermination` forces a program to wait for threads to complete execution
  - returns control to its caller either when all tasks executing in the `ExecutorService` complete or when the specified timeout elapses
  - If all tasks complete before `awaitTermination` times out, returns `true`; otherwise returns `false`

```java
// Fig. 23.7: SimpleArray.java
// Class that manages an integer array to be shared by multiple threads.
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
```

Track where next item will be placed

ray.java

```

      // 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()
   {
      String arrayString = "\nContents of SimpleArray:\n";

      for ( int i = 0; i < array.length; i++ )
         arrayString += array[ i ] + " ";

      return arrayString;
   } // end method toString
} // end class SimpleArray
```

Place the item

Specify where next item will be placed

Outline

```
// Fig. 23.8: 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
```

ArrayWriter.java

```
// Fig 23.9: 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 );

```

SharedArrayTest .java

Both `ArrayWriter`s share the same `SimpleArray` object

```
         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 wait for tasks to finish." );
      } // end catch
   } // end main
} // end class 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-1 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.

Outline

SharedArrayTest .java

(2 of 2)

23.5.2 Synchronized Data Sharing—Making Operations Atomic

- **Simulate atomicity by ensuring that only one thread carries out a set of operations at a time**
- **Immutable data shared across threads**
  - **declare the corresponding data fields `final` to indicate that variables' values will not change after they are initialized**

Software Engineering Observation 23.1

Place all accesses to mutable data that may be shared by multiple threads inside `synchronized` statements or `synchronized` methods that synchronize on the same lock. When performing multiple operations on shared data, hold the lock for the entirety of the operation to ensure that the operation is effectively atomic.

```java
// Fig. 23.10: SimpleArray.java
// Class that manages an integer array to be shared by multiple threads.
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 in

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

Using `synchronized` prevents more than one thread at a time from calling this method on a specific `SimpleArray` object

Outline

```

      // 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()
   {
      String arrayString = "\nContents of SimpleArray:\n";

      for ( int i = 0; i < array.length; i++ )
         arrayString += array[ i ] + " ";

      return arrayString;
   } // end method toString
} // end class SimpleArray
```

SimpleArray.java

(2 of 3)

```
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
```

Performance Tip 23.2

Keep the duration of `synchronized` statements as short as possible while maintaining the needed synchronization. This minimizes the wait time for blocked threads. Avoid performing I/O, lengthy calculations and operations that do not require synchronization with a lock held.

Good Programming Practice 23.1

Always declare data fields that you do not expect to change as `final`. Primitive variables that are declared as `final` can safely be shared across threads. An object reference that is declared as `final` ensures that the object it refers to will be fully constructed and initialized before it is used by the program and prevents the reference from pointing to another object.

23.6 Producer/Consumer Relationship without Synchronization

- Multithreaded producer/consumer relationship
  - Producer thread generates data and places it in a shared object called a buffer
  - Consumer thread reads data from the buffer
- Operations on the buffer data shared by a producer and a consumer are state dependent
  - Should proceed only if the buffer is in the correct state
  - If in a not-full state, the producer may produce
  - If in a not-empty state, the consumer may consume
- Must synchronize access to ensure that data is written to the buffer or read from the buffer only if the buffer is in the proper state

Outline

Buffer.java

```java
// Fig. 23.11: Buffer.java
// Buffer interface specifies me                                Consumer.
public interface Buffer
{
   // place int value into Buffe
   public void set( int value )

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

Interface `Buffer` that will be implemented in each of our Producer/Consumer examples

Outline

`Producer.java`

(1 of 2)

```java
// Fig. 23.12: Producer.java
// Producer with a run method that inserts the values 1 to 10 in buffer.
import java.util.Random;

public class Producer implements Runnable
{
   private final static Random generator = new Rand
   private final Buffer sharedLocation; // referenc

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

   // store values from 1 to 1
   public void run()
   {
      int sum = 0;

      for ( int count = 1; count <= 10; count++ )
      {
         try // sleep 0 to 3 seconds, then place value in Buffer
         {
            Thread.sleep( generator.nextInt( 3
            sharedLocation.set( count ); // se
            sum += count; // increment sum of v
            System.out.printf( "\t%2d\n", sum );
         } // end try
```

Class `Producer` represents a `Runnable` task that places values in a `Buffer`

Defines the `Producer`’s task

Places a value in the `Buffer`

```
         // if lines 25 or 26 get interrupted, print stack trace
         catch ( InterruptedException exception )
         {
            exception.printStackTrace();
         } // end catch
      } // end for

      System.out.println(
         "Producer done producing\nTerminating Producer" );
   } // end method run
} // end class Producer
```

Outline

Producer.java

(2 of 2)

```java
// Fig. 23.13: 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 Rand
   private final Buffer sharedLocation; // referenc

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

   // read sharedLocation's va                                s
   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( 3
            sum += sharedLocation.get();
            System.out.printf( "\t\t\t%2d\n", s
         } // end try
```

Class `Consumer` represents a `Runnable` task that reads values from a `Buffer`

Defines the `Consumer`'s task

Reads a value from the `Buffer`

```
         // 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
```

Outline

```
// Fig. Fig. 23.14: UnsynchronizedBuffer.java
// UnsynchronizedBuffer maintains the shared integer that is accessed by
// a producer thread and a consumer thread via methods set a
public class UnsynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consum

   // 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
```

Unsynchronized implementation of interface `Buffer`

hronized
Buffer.java

```java
// Fig. 23.15: 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
```

Outline

SharedBufferTest.java

(1 of 3)

`Producer` and `Consumer` share the same unsynchronized `Buffer`

Outline

SharedBufferTest .java

(2 of 3)

```
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  2        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
```

(continued on next slide…)

(continued from previous slide…)

```
Action              Value    Sum of Produced Sum of 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
```

reads -1 bad data

1 read again

1 read again

1 read again

1 read again

5 is lost

7 never read

8 never read

9 never read

10 never read

Outline

SharedBufferTest.java

(3 of 3)

23.7 Producer/Consumer Relationship: `ArrayBlockingQueue`

- `ArrayBlockingQueue` (package `java.util.concurrent`)
  - Good choice for implementing a shared buffer
  - Implements interface `BlockingQueue`, which extends interface `Queue` and declares methods `put` and `take`
  - Method `put` places an element 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
  - Stores shared data in an array
  - Array size specified as a constructor argument
  - Array is fixed in size

```java
// Fig. 23.16: BlockingBuffer.java
// Creates a synchronized buffer using an ArrayBlockingQueue.
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingBuffer implements Buffer
{
   private final ArrayBlockingQueue<Integer> 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
```

Synchronized implementation of interface `Buffer` that uses an `ArrayBlockingQueue` to implement the synchronization

ngBuffer

(1 of 2)

Creates a single element buffer of type `ArrayBlockingQueue`

```java

   // return value from buffer
   public int get() throws InterruptedException
   {
      int readValue = 0; // initialize value read from buffer

      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
```

Outline

```
// Fig. 23.17: BlockingBufferTest.java
// Two threads manipulating a blocking buffer.
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
```

BlockingBuffer Test.java

(1 of 2)

`Producer` and `Consumer` share the same synchronized `Buffer`

Outline

```
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
```

BlockingBuffer Test.java

(2 of 2)

23.8 Producer/Consumer Relationship with Synchronization

- **Can implement a shared using the `synchronized` keyword and `Object` methods `wait`, `notify` and `notifyAll`**
  - **can be used with conditions to make threads wait when they cannot perform their tasks**
- **A thread that cannot continue with its task until some condition is satisfied can call `Object` method `wait`**
  - **releases the monitor lock on the object**
  - **thread waits in the waiting state while the other threads try to enter the object's `synchronized` statement(s) or method(s)**
- **A thread that completes or satisfies the condition on which another thread may be waiting can call `Object` method `notify`**
  - **allows a waiting thread to transition to the *runnable* state**
  - **the thread that was transitioned can attempt to reacquire the monitor lock**
- **If a thread calls `notifyAll`, all the threads waiting for the monitor lock become eligible to reacquire the lock**

Common Programming Error 23.1

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`.

Error-Prevention Tip 23.1

It is a good practice to use `notifyAll` to notify *waiting* threads to become *runnable*. Doing so avoids the possibility that your program would forget about waiting threads, which would otherwise starve.

Outline

zed
Buffer.java

of 3)

```
// Fig. 23.18: SynchronizedBuffer.java
// Synchronizing access to shared data using Object
// methods wait and notify.
public class SynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consu
   private boolean occupied = false; // whether the buffer is occupied

   // place value into buffer
   public synchronized void set( int value )
   {
      // while there are no empty locations, place th
      while ( occupied )
      {
         // output thread information and buffer information, then wait
         System.out.println( "Producer tries to write." );
         displayState( "E                                     ;
         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                                    nable state
   } // end method set; re                                    er
```

Synchronized implementation of interface Buffer that uses monitors and monitor locks

Synchronized method that allows a Producer to write a new value only if the buffer is empty

Producer cannot write so it must wait

Notify waiting Consumer that a value is now available

```
   // return value from buffer
   public synchronized int get()
   {
      // while no data to read, place thr
      while ( !occupied )
      {
         // output thread information and buffer information, then wait
         System.out.println( "Consumer tries to read." );
         displayState( "B                                        );
         wait();
      } // end while

```

Synchronized method that allows a `Consumer` to read a value only if the buffer is full

`Consumer` cannot read so it must wait

Outline

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

         displayState( "Consumer reads " + buffer );

         notifyAll(); // tell                              nable 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
```

Notify waiting `Producer` that a value is now available

SynchronizedBuffer.java

(3 of 3)

Error-Prevention Tip 23.2

Always invoke method `wait` in a loop that tests the condition the task is waiting on. It is possible that a thread will reenter the *runnable* state (via a timed wait or another thread calling `notifyAll`) before the condition is satisfied. Testing the condition again ensures that the thread will not erroneously execute if it was notified early.

```java
// Fig. 23.19: 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 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` and `Consumer` share the same synchronized `Buffer`

```
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 empty. Consumer 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 empty. Consumer waits.              6              true
```

(continued on next slide…)

SharedBuffer Test2.java

(continued from previous slide…)

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 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
```

Outline

SharedBuffer Test2.java

(3 of 3)

23.9 Producer/Consumer Relationship: Bounded Buffers

- Cannot make assumptions about the relative speeds of concurrent
- Bounded buffer
  - Used to minimize the amount of waiting time for threads that share resources and operate at the same average speeds
  - Key is to provide the buffer with enough locations to handle the anticipated “extra” production
  - `ArrayBlockingQueue` is a bounded buffer that handles all of the synchronization details for you

Performance Tip 23.3

Even when using a bounded buffer, it is possible that a producer thread could fill the buffer, which would force the producer to wait until a consumer consumed a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, a consumer thread must wait until the producer produces another value. The key to using a bounded buffer is to optimize the buffer size to minimize the amount of thread wait time, while not wasting space..

Outline

CircularBuffer.java

(1 of 3)

```java
// Fig. 23.20: 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
   {
      // output thread information and buffer information, then wait;
      // while no empty locations, place thread in bloc
      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
```

Determine whether buffer is full

Specify next write position in buffer

Outline

CircularBuffer.java

(2 of 3)

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

Determine whether buffer is empty

Specify the next read location in the buffer

CircularBuffer .java


```java

      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
```

```java
// Fig. 23.21: 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` and `Consumer` share the same synchronized circular `Buffer`

```
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
```

(continued on next slide…)

CircularBufferTest.java

(2 of 5)

(continued from previous slide…)

```
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
```

(continued on next slide…)

CircularBuffer Test.java

(3 of 5)

(continued from previous slide…)

```
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

Consumer reads 6 (buffer cells occupied: 2)
buffer cells:    7    8    6
               ---- ---- ----
                 R        W

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

Producer writes 9 (buffer cells occupied: 2)
buffer cells:    7    8    9
               ---- ---- ----
                W     R
```

(continued on next slide…)

CircularBuffer Test.java

(4 of 5)

(continued from previous slide…)

```
Consumer reads 8 (buffer cells occupied: 1)
buffer cells:    7    8    9
               ---- ---- ----
                W         R

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

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


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


Consumer read values totaling: 55
Terminating Consumer
```

Outline

CircularBuffer Test.java

(5 of 5)

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- **Introduced in Java SE 5**
- **Give programmers 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 package `java.util.concurrent.locks`)**
- **Call `Lock`'s `lock` method to acquire the lock**
  - **Once obtained by one thread, the `Lock` object will not allow another thread to obtain the `Lock` until the first thread releases the `Lock`**
- **Call `Lock`'s `unlock` method to release the lock**
- **All other threads attempting to obtain that `Lock` on a locked object are placed in the *waiting* state**

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- Class `ReentrantLock` (of package `java.util.concurrent.locks`) is a basic implementation of the `Lock` interface.
- `ReentrantLock` constructor takes a `boolean` argument that specifies whether the lock has a fairness policy
  - If `true`, the `ReentrantLock`'s fairness policy is "the longest-waiting thread will acquire the lock when it is available"—prevents starvation
  - If `false`, there is no guarantee as to which waiting thread will acquire the lock when it is available
- A thread that owns a `Lock` and determines that it cannot continue with its task until some condition is satisfied can wait on a condition object
- `Lock` objects allow you to explicitly declare the condition objects on which a thread may need to wait

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- `Condition` objects
  - Associated with a specific `Lock`
  - Created by calling a `Lock`'s `newCondition` method
- To wait on a `Condition` object, call the `Condition` 's `await` method
  - immediately releases the associated `Lock` and places the thread in the *waiting* state for that `Condition`
- Another thread can call `Condition` method `signal` to allow a thread in that `Condition`'s *waiting* state to return to the *runnable* state
  - Default implementation of `Condition` signals the longest-waiting thread
- `Condition` method `signalAll` transitions all the threads waiting for that condition to the *runnable* state
- When finished with a shared object, thread must call `unlock` to release the `Lock`

23.10 Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- `Lock` and `Condition` may be preferable to using the `synchronized` keyword
  - `Lock` objects allow you to interrupt waiting threads or to specify a timeout for waiting to acquire a lock
  - `Lock` object is not constrained to be acquired and released in the same block of code
- `Condition` objects can be used to specify multiple conditions on which threads may wait
  - Possible to indicate to waiting threads that a specific condition object is now true

Software Engineering Observation 23.2

Using a `ReentrantLock` with a fairness policy avoids indefinite postponement.

Performance Tip 23.4

Using a `ReentrantLock` with a fairness policy can decrease program performance significantly.

Common Programming Error 23.2

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. The two threads are waiting for each other, so the actions that would enable each thread to continue execution can never occur.

Error-Prevention Tip 23.3

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 object, a separate thread eventually will call `Condition` method `signal` to transition the thread waiting on the condition object back to the *runnable* state. If multiple threads may be waiting on the condition object, 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, starvation might occur.

Common Programming Error 23.3

An `IllegalMonitorStateException` occurs if a thread issues an `await`, a `signal`, or a `signalAll` on a condition object without having acquired the lock for that condition object.

Outline

Synchronized

va

```java
// Fig. 23.22: 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 informa
      try
      {
```

Synchronized implementation of interface `Buffer` that uses `Lock`s and `Condition`s

`Condition` indicating

`Condition` indicating when a consumer can read

Manually acquire the lock to implement mutual exclusion

Outline

```
         // 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 thread waiting t
         canRead.signal();
      } // end try
      finally
      {
         accessLock.unlock(); // unloc
      } // end finally
   } // end method set

```

Producer must wait until buffer is empty and release the lock

(2 of 4)

Producer signals the consumer that a value is available for reading

Release the lock so consumer can read

Outline

SynchronizedBuffer.java

(3 of 4)

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

      // output thread informati                              en wait
      try
      {
         // while 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 thread waiting f
         canWrite.signal();
      } // end try
```

Manually acquire the lock to implement mutual exclusion

Consumer must wait until buffer is full and release the lock

Consumer signals the producer that space is available for writing

```
      finally
      {
         accessLock.unlock(); // unloc
      } // 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
```

Release the lock so producer can write

SynchronizedBuffer.java

(4 of 4)

Error-Prevention Tip 23.4

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

Common Programming Error 23.4

Forgetting to `signal` a waiting thread is a logic error. The thread will remain in the *waiting* state, which will prevent the thread from proceeding. Such waiting can lead to indefinite postponement or deadlock.

```java
// Fig. 23.23: 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
```

```
Operation                               Buffer          Occupied
---------                               ------          --------

Producer writes 1                       1               true

Producer tries to write.
Buffer full. Producer waits.            1               true
```

(continued on next slide…)

SharedBuffer
Test2.java

(continued from prevoius slide…)

```
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.         5               false

Producer writes 5                     5               true

Consumer reads 5                      5               false
```

(continued on next slide…)

Outline

SharedBuffer Test2.java

(2 of 3)

(continued from previous slide…)

```
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
```

23.11 Multithreading with GUI

- Event dispatch thread handles interactions with the application's GUI components
  - All tasks that interact with an application's GUI are placed in an event queue
  - Executed sequentially by the event dispatch thread
- Swing GUI components are not thread safe
  - Thread safety achieved by ensuring that Swing components are accessed from only the event dispatch thread—known as thread confinement
- Preferable to handle long-running computations in a separate thread, so the event dispatch thread can continue managing other GUI interactions
- Class `SwingWorker` (in package `javax.swing`) implements interface `Runnable`
  - Performs long-running computations in a worker thread
  - Updates Swing components from the event dispatch thread based on the computations' results

Method	Description
`doInBackground`	Defines a long computation and is called in a worker thread.
`done`	Executes on the event dispatch thread when `doInBackground` returns.
`execute`	Schedules the `SwingWorker` object to be executed in a worker thread.
`get`	Waits for the computation to complete, then returns the result of the computation (i.e., the return value of `doInBackground`).
`publish`	Sends intermediate results from the `doInBackground` method to the process method for processing on the event dispatch thread.
`process`	Receives intermediate results from the `publish` method and processes these results on the event dispatch thread.
`setProgress`	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

Fig. 23.24 | Commonly used `SwingWorker` methods.

23.11.1 Performing Computations in a Worker Thread

- To use a `SwingWorker`
  - Extend `SwingWorker`
  - Overrides methods `doInBackground` and `done`
  - `doInBackground` performs the computation and returns the result
  - `done` displays the results in the GUI after `doInBackground` returns
- `SwingWorker` is a generic class
  - First type parameter indicates the type returned by `doInBackground`
  - Second indicates the type passed between the `publish` and `process` methods to handle intermediate results
- `ExecutionException` thrown if an exception occurs during the computation

```java
// Fig. 23.25: BackgroundCalculator.java
// SwingWorker subclass for calculating Fibonacci numbers
// in a background thread.
import javax.swing.SwingWorker;
import javax.swing.JLabel;
import java.util.concurrent.ExecutionException;

public class BackgroundCalculator extends SwingWorker< String, Object >
{
   private final int n; // Fibonacci number to calculate
   private final JLabel resultJLabel; // JLabel to display the result

   // constructor
   public BackgroundCalculator( int number, JLabel label )
   {
      n = number;
      resultJLabel = label;
   } // end BackgroundCalculator constructor

   // long-running code to be run in a worker thread
   public String doInBackground()
   {
      long nthFib = fibonacci( n );
      return String.valueOf( nthFib );
   } // end method doInBackground

```

Create a subclass of SwingWorker to

Possibly lengthy Fibonacci calculation to perform in the background

(1 of 2)

Outline

BackgroundCalculator.java

(2 of 2)

```
   // code to run on the event dispatch thread when doInBackground returns
   protected void done()
   {
      try
      {
         // get the result of doInBackground and
         resultJLabel.setText( get() );
      } // end try
      catch ( InterruptedException ex )
      {
         resultJLabel.setText( "Interrupted while waiting for results." );
      } // end catch
      catch ( ExecutionException ex )
      {
         resultJLabel.setText(
            "Error encountered while performing calculation." );
      } // end catch
   } // end method done

   // recursive method fibonacci; calculates nth Fibonacci number
   public long fibonacci( long number )
   {
      if ( number == 0 || number == 1 )
         return number;
      else
         return fibonacci( number - 1 ) + fibonacci( number - 2 );
   } // end method fibonacci
} // end class BackgroundCalculator
```

Display the calculation results when done

Software Engineering Observation 23.3

Any GUI components that will be manipulated by `SwingWorker` methods, such as components that will be updated from methods `process` or `done`, should be passed to the `SwingWorker` subclass's constructor and stored in the subclass object. This gives these methods access to the GUI components they will manipulate.

Outline

FibonacciNumbers.java

(1 of 5)

```
// Fig. 23.26: FibonacciNumbers.java
// Using SwingWorker to perform a long calculation with
// intermediate results displayed in a GUI.
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.border.TitledBorder;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.util.concurrent.ExecutionException;

public class FibonacciNumbers extends JFrame
{
   // components for calculating the Fibonacci of a user-entered number
   private final JPanel workerJPanel =
      new JPanel( new GridLayout( 2, 2, 5, 5 ) );
   private final JTextField numberJTextField = new JTextField();
   private final JButton goJButton = new JButton( "Go" );
   private final JLabel fibonacciJLabel = new JLabel();

   // components and variables for getting the next Fibonacci number
   private final JPanel eventThreadJPanel =
      new JPanel( new GridLayout( 2, 2, 5, 5 ) );
   private int n1 = 0; // initialize with first Fibonacci number
```

```
   private int n2 = 1; // initialize with second Fibonacci number
   private int count = 1;
   private final JLabel nJLabel = new JLabel( "Fibonacci of 1: " );
   private final JLabel nFibonacciJLabel =
      new JLabel( String.valueOf( n2 ) );
   private final JButton nextNumberJButton = new JButton( "Next Number" );

   // constructor
   public FibonacciNumbers()
   {
      super( "Fibonacci Numbers" );
      setLayout( new GridLayout( 2, 1, 10, 10 ) );

      // add GUI components to the SwingWorker panel
      workerJPanel.setBorder( new TitledBorder(
         new LineBorder( Color.BLACK ), "With SwingWorker" ) );
      workerJPanel.add( new JLabel( "Get Fibonacci of:" ) );
      workerJPanel.add( numberJTextField );
      goJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent event )
            {
               int n;

```

Outline

FibonacciNumbers.java

(3 of 5)

```
            try
            {
               // retrieve user's input as an integer
               n = Integer.parseInt( numberJTextField.getText() );
            } // end try
            catch( NumberFormatException ex )
            {
               // display an error message if the user did not
               // enter an integer
               fibonacciJLabel.setText( "Enter an integer." );
               return;
            } // end catch

            // indicate that the calculation has begun
            fibonacciJLabel.setText( "Calculating..." );

            // create a task to perform calculation in background
            BackgroundCalculator task =
               new BackgroundCalculator( n, fibonacciJLabel );
            task.execute(); // execute the task
         } // end method actionPerformed
      } // end anonymous inner class
   ); // end call to addActionListener
   workerJPanel.add( goJButton );
   workerJPanel.add( fibonacciJLabel );

```

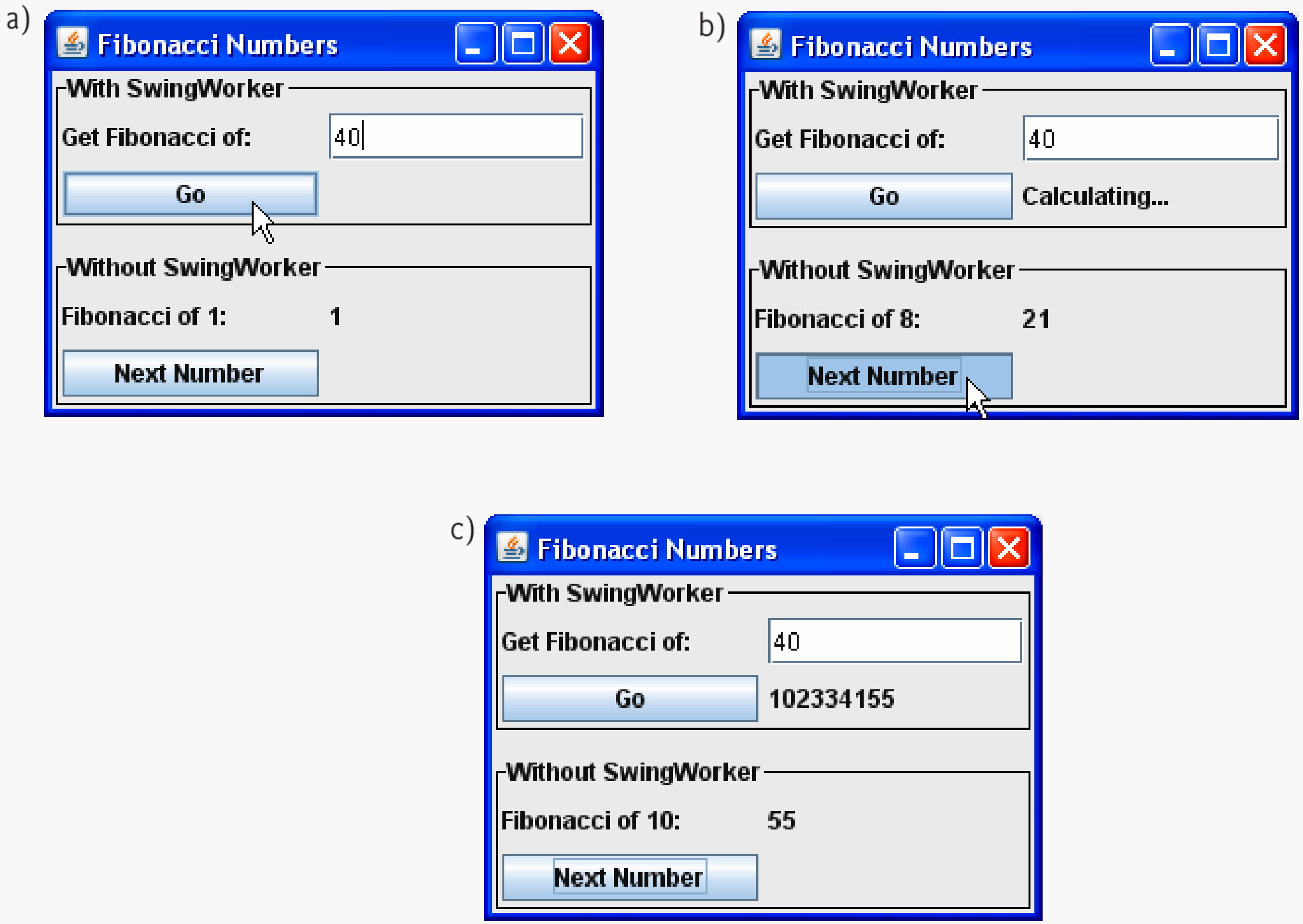
Outline

FibonacciNumbers.java

(4 of 5)

```
      // add GUI components to the event-dispatching thread panel
      eventThreadJPanel.setBorder( new TitledBorder(
         new LineBorder( Color.BLACK ), "Without SwingWorker" ) );
      eventThreadJPanel.add( nJLabel );
      eventThreadJPanel.add( nFibonacciJLabel );
      nextNumberJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent event )
            {
               // calculate the Fibonacci number after n2
               int temp = n1 + n2;
               n1 = n2;
               n2 = temp;
               ++count;

               // display the next Fibonacci number
               nJLabel.setText( "Fibonacci of " + count + ": " );
               nFibonacciJLabel.setText( String.valueOf( n2 ) );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
      eventThreadJPanel.add( nextNumberJButton );

      add( workerJPanel );
      add( eventThreadJPanel );
      setSize( 275, 200 );
      setVisible( true );
   } // end constructor

```

```
   // main method begins program execution
   public static void main( String[] args )
   {
      FibonacciNumbers application = new FibonacciNumbers();
      application.setDefaultCloseOperation( EXIT_ON_CLOSE );
   } // end main
} // end class FibonacciNumbers
```



23.11.2 Processing Intermediate Results with `SwingWorker`

- `SwingWorker` methods
  - `publish` repeatedly sends intermediate results to method `process`
  - `process` executes in the event dispatch thread and receives data from method `publish` then displays the data in a GUI component
  - `setProgress` updates the progress property
- Values are passed asynchronously between `publish` in the worker thread and `process` in the event dispatch thread
- `process` is not necessarily invoked for every call to `publish`
- `PropertyChangeListener`
  - Interface from package `java.beans`
  - Defines method `propertyChange`
  - Each call to `setProgress` generates a `PropertyChangeEvent` to indicate that the progress property has changed

Outline

PrimeCalculator .java

(1 of 5)

```java
// Fig. 23.27: PrimeCalculator.java
// Calculates the first n primes, displaying them as they are found.
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.SwingWorker;
import java.util.Random;
import java.util.List;
import java.util.concurrent.ExecutionException;

public class PrimeCalculator extends SwingWorker< Integer, Integer >
{
   private final Random generator = new Random();
   private final JTextArea intermediateJTextArea; // displays found primes
   private final JButton getPrimesJButton;
   private final JButton cancelJButton;
   private final JLabel statusJLabel; // displays status of calculation
   private final boolean primes[]; // boolean array for finding primes
   private boolean stopped = false; // flag indicating cancelation

   // constructor
   public PrimeCalculator( int max, JTextArea intermediate, JLabel status,
      JButton getPrimes, JButton cancel )
   {
      intermediateJTextArea = intermediate;
      statusJLabel = status;
      getPrimesJButton = getPrimes;
      cancelJButton = cancel;
      primes = new boolean[ max ];

```

```
      // initialize all primes array values to true
      for ( int i = 0; i < max; i ++ )
         primes[ i ] = true;
   } // end constructor

   // finds all primes up to max using the Sieve of Eratosthenes
   public Integer doInBackground()
   {
      int count = 0; // the number of primes found

      // starting at the third value, cycle through the array and put
      // false as the value of any greater number that is a multiple
      for ( int i = 2; i < primes.length; i++ )
      {
         if ( stopped ) // if calculation has been canceled
            return count;
         else
         {
            setProgress( 100 * ( i + 1 ) / primes.length );

            try
            {
               Thread.currentThread().sleep( generator.nextInt( 5 ) );
            } // end try
            catch ( InterruptedException ex )
            {
               statusJLabel.setText( "Worker thread interrupted" );
               return count;
            } // end catch

```

Specify progress status as a percentage of the number of primes we are calculating

Outline

```
            if ( primes[ i ] ) // i is prime
            {
               publish( i ); // make                      rime list
               ++count;

               for ( int j = i + i; j < primes.length; j += i )
                  primes[ j ] = false; // i is not prime
            } // end if
         } // end else
      } // end for

      return count;
   } // end method doInBackground

   // displays published values in primes list
   protected void process( List< Integer > publishedVals )
   {
      for ( int i = 0; i < publishedVals.size(); i++ )
         intermediateJTextArea.append( publishedVals.get( i ) + "\n" );
   } // end method process
```

Publish each prime as it is discovered

Process all the published prime values

PrimeCalculator .java

(3 of 5)

```java

   // code to execute when doInBackground completes
   protected void done()
   {
      getPrimesJButton.setEnabled( true ); // enable Get Primes button
      cancelJButton.setEnabled( false ); // disable Cancel button

      int numPrimes;

      try
      {
         numPrimes = get(); // retrieve doInBackground return value
      } // end try
      catch ( InterruptedException ex )
      {
         statusJLabel.setText( "Interrupted while waiting for results." );
         return;
      } // end catch
      catch ( ExecutionException ex )
      {
         statusJLabel.setText( "Error performing computation." );
         return;
      } // end catch
```

Outline

```
        statusJLabel.setText( "Found " + numPrimes + " primes." );
    } // end method done

    // sets flag to stop looking for primes
    public void stopCalculation()
    {
        stopped = true;
    } // end method stopCalculation
} // end class PrimeCalculator
```

PrimeCalculator.java

(5 of 5)

Outline

FindPrimes.java

(1 of 6)

```java
// Fig 23.28: FindPrimes.java
// Using a SwingWorker to display prime numbers and update a JProgressBar
// while the prime numbers are being calculated.
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import javax.swing.JButton;
import javax.swing.JProgressBar;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.ScrollPaneConstants;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.concurrent.ExecutionException;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

public class FindPrimes extends JFrame
{
   private final JTextField highestPrimeJTextField = new JTextField();
   private final JButton getPrimesJButton = new JButton( "Get Primes" );
   private final JTextArea displayPrimesJTextArea = new JTextArea();
   private final JButton cancelJButton = new JButton( "Cancel" );
   private final JProgressBar progressJProgressBar = new JProgressBar();
   private final JLabel statusJLabel = new JLabel();
   private PrimeCalculator calculator;

```

```
   // constructor
   public FindPrimes()
   {
      super( "Finding Primes with SwingWorker" );
      setLayout( new BorderLayout() );

      // initialize panel to get a number from the user
      JPanel northJPanel = new JPanel();
      northJPanel.add( new JLabel( "Find primes less than: " ) );
      highestPrimeJTextField.setColumns( 5 );
      northJPanel.add( highestPrimeJTextField );
      getPrimesJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent e )
            {
               progressJProgressBar.setValue( 0 ); // reset JProgressBar
               displayPrimesJTextArea.setText( "" ); // clear JTextArea
               statusJLabel.setText( "" ); // clear JLabel

               int number;

               try
               {
                  // get user input
                  number = Integer.parseInt(
                     highestPrimeJTextField.getText() );
               } // end try
```


FindPrimes.java

```java
        catch ( NumberFormatException ex )
        {
           statusJLabel.setText( "Enter an integer." );
           return;
        } // end catch

        // construct a new PrimeCalculator object
        calculator = new PrimeCalculator( number,
           displayPrimesJTextArea, statusJLabel, getPrimesJButton,
           cancelJButton );

        // listen for progress bar property changes
        calculator.addPropertyChangeListener(
           new PropertyChangeListener()
           {
              public void propertyChange( PropertyChangeEvent e )
              {
                 // if the changed property is progress,
                 // update the progress bar
                 if ( e.getPropertyName().equals( "progress" ) )
                 {
                    int newValue = ( Integer ) e.getNewValue();
                    progressJProgressBar.setValue( newValue );
                 } // end if
              } // end method propertyChange
           } // end anonymous inner class
        ); // end call to addPropertyChangeListener
```

Outline

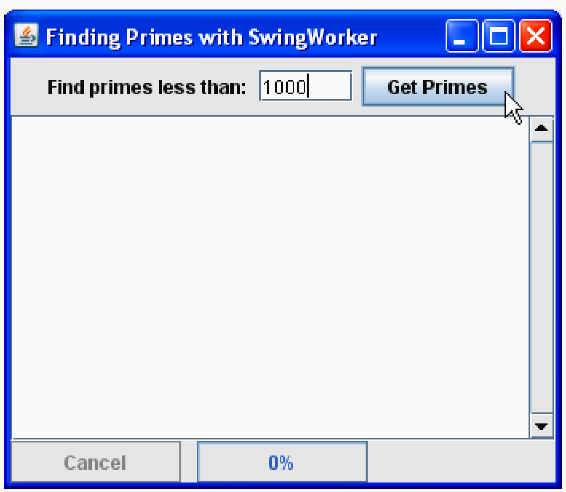
FindPrimes.java

(3 of 6)

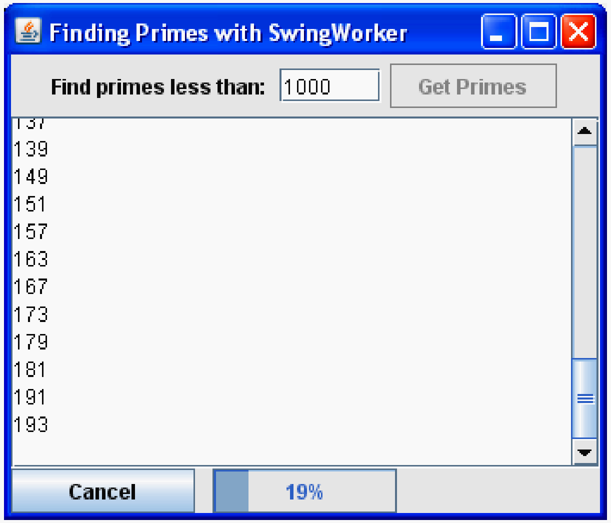
```

               // disable Get Primes button and enable Cancel button
               getPrimesJButton.setEnabled( false );
               cancelJButton.setEnabled( true );

               calculator.execute(); // execute the PrimeCalculator object
            } // end method ActionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
      northJPanel.add( getPrimesJButton );

      // add a scrollable JList to display results of calculation
      displayPrimesJTextArea.setEditable( false );
      add( new JScrollPane( displayPrimesJTextArea,
         ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
         ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER ) );

      // initialize a panel to display cancelJButton,
      // progressJProgressBar, and statusJLabel
      JPanel southJPanel = new JPanel( new GridLayout( 1, 3, 10, 10 ) );
      cancelJButton.setEnabled( false );
      cancelJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent e )
            {
               calculator.stopCalculation(); // cancel the calculation
            } // end method ActionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
```

```java
      southJPanel.add( cancelJButton );
      progressJProgressBar.setStringPainted( true );
      southJPanel.add( progressJProgressBar );
      southJPanel.add( statusJLabel );

      add( northJPanel, BorderLayout.NORTH );
      add( southJPanel, BorderLayout.SOUTH );
      setSize( 350, 300 );
      setVisible( true );
   } // end constructor

   // main method begins program execution
   public static void main( String[] args )
   {
      FindPrimes application = new FindPrimes();
      application.setDefaultCloseOperation( EXIT_ON_CLOSE );
   } // end main
} // end class FindPrimes
```

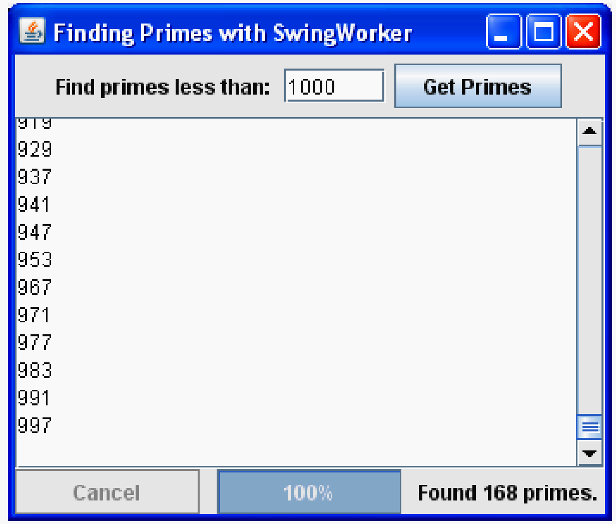
a)



b)



c)



Outline

`FindPrimes.java`

(6 of 6)

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

- `Callable` interface
  - package `java.util.concurrent`
  - declares a single method named `call`
  - similar to `Runnable`, but method `call` allows the thread to return a value or to throw a checked exception
- `ExecutorService` method `submit` executes a `Callable`
  - Returns an object of type `Future` (of package `java.util.concurrent`) that represents the executing `Callable`
  - `Future` declares method `get` to return the result of the `Callable` and other methods to manage a `Callable`'s execution