

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.



- 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



- 23.8 Producer/Consumer Relationship with Synchronization
- 23.9 Producer/Consumer Relationship: Bounded Buffers
- 23.10 Producer/Consumer Relationship: The ~~Lock~~ and ~~Condition~~ Interfaces
- 23.11 Multithreading with GUI
- 23.12 Other Classes and Interfaces in ~~java.util.concurrent~~
  - 23.11.1 Performing Computations in a Worker Thread
  - 23.11.2 Processing Intermediate Results with ~~SwingWorker~~
- 23.13 Wrap-Up



## 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 ~~java.util.concurrent~~ 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

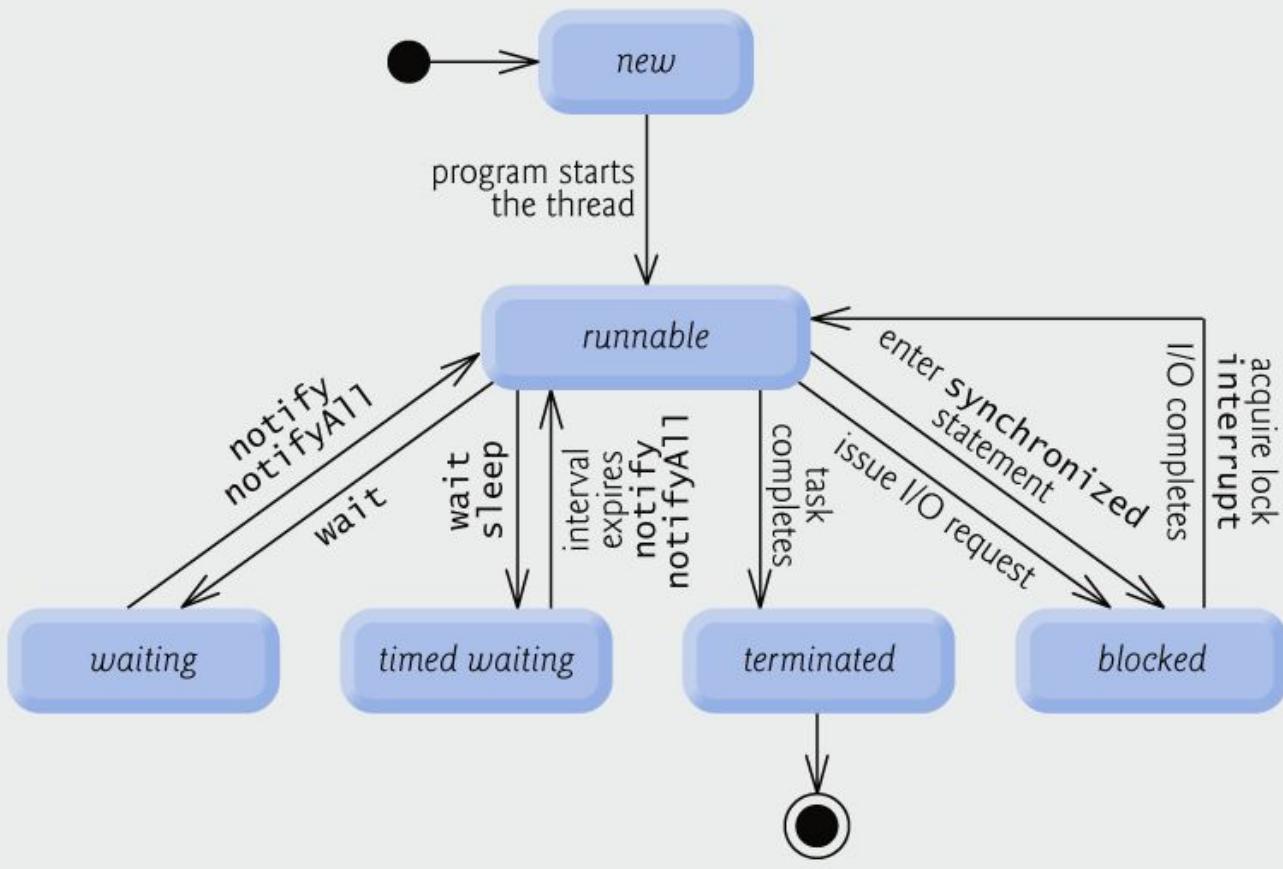
- The **Lock** and **Condition** interfaces should be used only by advanced programmers who are familiar with the common traps and pitfalls of concurrent programming



## 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

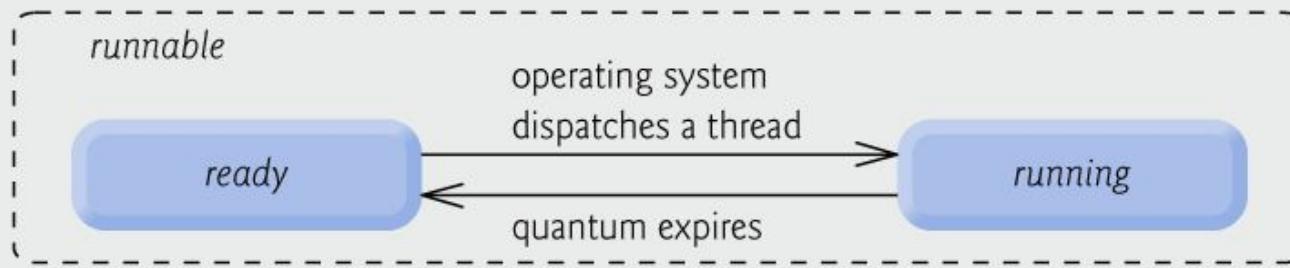
- ***Timed waiting* and *waiting* threads cannot use a processor, even if one is available.**
- **A *Runnable* thread can transition to the *timed waiting* state if it provides an optional wait interval when it is waiting for another thread to perform a task.**
  - returns to the *Runnable* state when
    - 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)

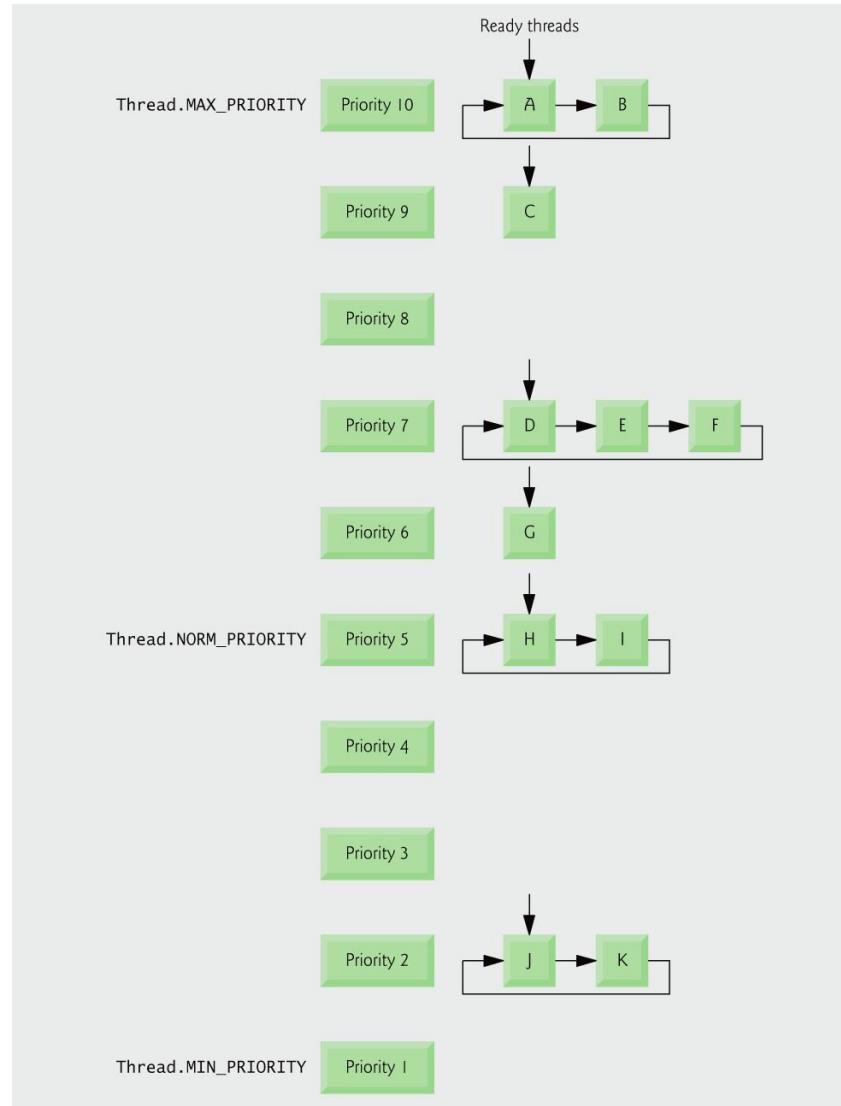


## Portability Tip 23.1

---

**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 Runnable s 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



```
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 public class PrintTask implements Runnable {
6
7     private final int sleepTime; // random sleep time for thread
8     private final String taskName; // name of task
9     private final static Random generator = new Random();
10
11    public PrintTask( String name )
12    {
13        taskName = name; // set task name
14
15        // pick random sleep time between 0 and 5 seconds
16        sleepTime = generator.nextInt( 5000 ); // milliseconds
17    } // end PrintTask constructor
18
```

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

## Outline

PrintTask.java  
Java

(1 of 2 )



```
19 // method run contains the code to be executed
20 public void run() ← Define task in method xrun
21 {
22     try // put thread to sleep for sleepTime amount of time
23     {
24         System.out.printf( "%s going to sleep for %d milliseconds.\n",
25             taskName, sleepTime );
26         Thread.sleep( sleepTime ); // put thread to sleep
27     } // end try
28     catch ( InterruptedException exception )
29     {
30         System.out.printf( "%s %s\n",
31             taskName,
32             "terminated prematurely due to interruption" );
33     } // end catch
34
35     // print task name
36     System.out.printf( "%s done sleeping\n", taskName );
37 } // end method run
38 } // end class PrintTask
```

## Outline

### **PrintTask.java**

(2 of 2 )



## Outline

### **ThreadCreator top.java**

(1 of 2 )

```

1 // Fig. 23.5: ThreadCreator.java
2 // Creating and starting three threads to execute Runnables.
3 import java.lang.Thread;
4
5 public class ThreadCreator
6 {
7     public static void main( String[] args )
8     {
9         System.out.println( "Creating threads" );
10
11     // create each thread with a new targeted runnable
12     Thread thread1 = new Thread( new PrintTask( "task1" ) );
13     Thread thread2 = new Thread( new PrintTask( "task2" ) );
14     Thread thread3 = new Thread( new PrintTask( "task3" ) );
15
16     System.out.println( "Threads created, starting tasks." );
17
18     // start threads and place in runnable state
19     thread1.start(); // invokes task1's run method
20     thread2.start(); // invokes task2's run method
21     thread3.start(); // invokes task3's run method
22
23     System.out.println( "Tasks started, main ends. \n" );
24 } // end main
25 } // end class RunnableTester

```

Create **Threads** to execute each new **Runnable** object

Start the **Threads** to begin processing the concurrent tasks



## Outline

### Threads top.java

(2 of 2 )

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
```



## 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 **Runnables**
- **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



```

1 // Fig. 23.6: TaskExecutor.java
2 // Using an ExecutorService to execute Runnables.
3 import java.util.concurrent.Executors;
4 import java.util.concurrent.ExecutorService;
5
6 public class TaskExecutor
7 {
8     public static void main( String[] args )
9     {
10         // create and name each runnable
11         PrintTask task1 = new PrintTask( "task1" );
12         PrintTask task2 = new PrintTask( "task2" );
13         PrintTask task3 = new PrintTask( "task3" );
14
15         System.out.println( "Starting Executor" );
16
17         // create ExecutorService to manage threads
18         ExecutorService threadExecutor = Executors.newCachedThreadPool();

```

## Outline

### **TaskExecutor OR.java**

(1 of 2 )

Creates an  
**EXECUTORS  
ERVICE** that  
manages a cached  
thread pool



```

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 // shut down worker threads when th
26 threadExecutor.shutdown(); ←
27
28 System.out.println( "Tasks started,
29 } // end main
30 } // end class TaskExecutor

```

Use the **ExecutorService**'s **execute** method to assign new **Runnable** object  
read in the thread pool  
**or.java**

(2 of 2 )

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

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**



## Outline

### SimpleArray.java

(1 of 2 )

```

1 // Fig. 23.7: SimpleArray.java
2 // Class that manages an integer array to be shared by multiple threads.
3 import java.util.Random
4
5 public class SimpleArray // CAUTION: NOT THREAD SAFE!
6 {
7     private final int array[]; // the shared integer array
8     private int writeIndex = 0; // index of next element to be written
9     private final static Random generator = new Random();
10
11    // construct a SimpleArray of a given size
12    public SimpleArray( int size )
13    {
14        array = new int[ size ];
15    } // end constructor
16
17    // add a value to the shared array
18    public void add( int value )
19    {
20        int position = writeIndex; // store the write index ←
21
22        try
23        {
24            // put thread to sleep for 0-499 milliseconds
25            Thread.sleep( generator.nextInt( 500 ) );
26        } // end try
27        catch ( InterruptedException ex )
28        {
29            ex.printStackTrace();
30        } // end catch

```

Track where next item will  
be placed



```

31
32 // put value in the appropriate element
33 array[ position ] = value;
34 System.out.printf( "%s wrote %d to element %d.\n",
35     Thread.currentThread().getName(), value, position );
36
37 ++writeIndex; // increment index of element to be written next
38 System.out.printf( "Next write index: %d\n", writeIndex );
39 } // end method add
40
41 // used for outputting the contents of the shared integer array
42 public String toString()
43 {
44     String arrayString = "\nContents of SimpleArray:\n";
45
46     for ( int i = 0; i < array.length; i++ )
47         arrayString += array[ i ] + " ";
48
49     return arrayString;
50 } // end method toString
51 } // end class SimpleArray

```

Place the item

Outline

Specify where  
next item will be  
placed

CLIPPER

(- or -)



```
1 // Fig. 23.8: ArrayWriter.java
2 // Adds integers to an array shared with other Runnables
3 import java.lang.Runnable;
4
5 public class ArrayWriter implements Runnable
6 {
7     private final SimpleArray sharedSimpleArray;
8     private final int startValue;
9
10    public ArrayWriter( int value, SimpleArray array )
11    {
12        startValue = value;
13        sharedSimpleArray = array;
14    } // end constructor
15
16    public void run()
17    {
18        for ( int i = startValue; i < startValue + 3; i++ )
19        {
20            sharedSimpleArray.add( i ); // add an element to the shared array
21        } // end for
22    } // end method run
23 } // end class ArrayWriter
```

## Outline

### arraywriter java



```

1 // Fig. 23.9: SharedArrayTest.java
2 // Executes two Runnable s to add elements to a shared SimpleArray.
3 import java.util.concurrent.Executors;
4 import java.util.concurrent.ExecutorService;
5 import java.util.concurrent.TimeUnit;
6
7 public class SharedArrayTest
8 {
9     public static void main( String[] args )
10    {
11        // construct the shared object
12        SimpleArray sharedSimpleArray = new SimpleArray( 6 );
13
14        // create two tasks to write to the shared SimpleArray
15        ArrayWriter writer1 = new ArrayWriter( 1, sharedSimpleArray );
16        ArrayWriter writer2 = new ArrayWriter( 11, sharedSimpleArray );
17
18        // execute the tasks with an ExecutorService
19        ExecutorService executor = Executors.newCachedThreadPool();
20        executor.execute( writer1 );
21        executor.execute( writer2 );
22
23        executor.shutdown();
24
25        try
26        {
27            // wait 1 minute for both writers to finish executing
28            boolean tasksEnded = executor.awaitTermination(
29                1, TimeUnit.MILLISECONDS );
30

```

## Outline

### SharedArrayTest.java

(1 of 2 )

Both  
**ArrayWriter**  
**P**s share the same  
**SimpleArra**  
**y** object



```

31     if ( tasksEnded )
32         System.out.println( sharedSimpleArray ); // print contents
33     else
34         System.out.println(
35             "Timed out while waiting for tasks to finish." );
36     } // end try
37     catch ( InterruptedException ex )
38     {
39         System.out.println(
40             "Interrupted while wait for tasks to finish." );
41     } // end catch
42 } // end main
43 } // 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-2 wrote 11 to element 0.
Next write index: 4
pool-1-thread-2 wrote 12 to element 4.
Next write index: 5
pool-1-thread-2 wrote 13 to element 5.
Next write index: 6

```

Contents of SimpleArray:  
11 2 3 0 12 13

## Outline

### SharedArrayTest.java

(2 of 2)

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.



## 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.



```

1 // Fig. 23.10: SimpleArray.java
2 // Class that manages an integer array to be shared by multiple threads.
3 import java.util.Random
4
5 public class SimpleArray
6 {
7     private final int array[]; // the shared integer array
8     private int writeIndex = 0; // index of next element to be written
9     private final static Random generator = new Random();
10
11    // construct a SimpleArray of a given size
12    public SimpleArray( int size )
13    {
14        array = new int[ size ];
15    } // end constructor
16
17    // add a value to the shared array
18    public synchronized void add( int value ) ←
19    {
20        int position = writeIndex; // store the write index
21
22        try
23        {
24            // put thread to sleep for 0-499 milliseconds
25            Thread.sleep( generator.nextInt( 500 ) );
26        } // end try
27        catch ( InterruptedException ex )
28        {
29            ex.printStackTrace();
30        } // end catch

```

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

## Outline

### SimpleArray.java

(1 of 3)



```
31
32     // put value in the appropriate element
33     array[ position ] = value;
34     System.out.printf( "%s wrote %d to element %d.\n",
35                         Thread.currentThread().getName(), value, position );
36
37     ++writeIndex; // increment index of element to be written next
38     System.out.printf( "Next write index: %d\n", writeIndex );
39 } // end method add
40
41 // used for outputting the contents of the shared integer array
42 public String toString()
43 {
44     String arrayString = "\nContents of SimpleArray:\n";
45
46     for ( int i = 0; i < array.length; i++ )
47         arrayString += array[ i ] + " ";
48
49     return arrayString;
50 } // end method toString
51 } // end class SimpleArray
```

## Outline

### 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

## Outline

### **SimpleArray.java**

(3 of 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**



```
1 // Fig. 23.11: Buffer.java
2 // Buffer interface specifies methods for producer and consumer.
3 public interface Buffer {
4 {
5     // place int value into Buffer
6     public void set( int value )
7
8     // return int value from Buffer
9     public int get() throws InterruptedException;
10 } // end interface Buffer
```

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

## Outline

### **Buffer.java**



```

1 // Fig. 23.12: Producer.java
2 // Producer with a run method that inserts the values 1 to 10 in buffer.
3 import java.util.Random;
4
5 public class Producer implements Runnable {
6
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // reference
9
10    // constructor
11    public Producer( Buffer shared ) {
12
13        sharedLocation = shared;
14    } // end Producer constructor
15
16    // store values from 1 to 10
17    public void run() {
18
19        int sum = 0;
20
21        for ( int count = 1; count <= 10; count++ )
22        {
23            try // sleep 0 to 3 seconds, then place value in Buffer
24            {
25                Thread.sleep( generator.nextInt( 3 ) );
26                sharedLocation.set( count ); // set value
27                sum += count; // increment sum of values
28                System.out.printf( "\t%d\n", sum );
29            } // end try

```

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

## Outline

### Producer.java

(1 of 2)

Defines the  
**Producer**'s task

Places a value in the  
**Buffer**



```
30 // if lines 25 or 26 get interrupted, print stack trace
31 catch ( InterruptedException exception )
32 {
33     exception.printStackTrace();
34 } // end catch
35 } // end for
36
37 System.out.println(
38     "Producer done producing\nTerminating Producer" );
39 } // end method run
40 } // end class Producer
```

## Outline

### Producer.java

(2 of 2 )



```

1 // Fig. 23.13: Consumer.java
2 // Consumer with a run method that loops, reading 10 values from buffer.
3 import java.util.Random;
4
5 public class Consumer implements Runnable {
6
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // referenced by Buffer
9
10    // constructor
11    public Consumer( Buffer shared ) {
12
13        sharedLocation = shared;
14    } // end Consumer constructor
15
16    // read sharedLocation's value
17    public void run() {
18
19        int sum = 0;
20
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( 3 ) );
27                sum += sharedLocation.get(); // reads value from Buffer
28                System.out.printf( "\t\t%d\n", sum );
29            } // end try

```

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

## Outline

### Consumer.java

(1 of 2)

Defines the  
**consumer**'s task

Reads a value from  
the **Buffer**



```
30 // if lines 26 or 27 get 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 %d\n%s\n",
38     "Consumer read values totaling", sum, "Terminating Consumer" );
39 } // end method run
40 } // end class Consumer
```

## Outline

### **Consumer.java**

(2 of 2 )



```

1 // Fig. Fig. 23.14: UnsynchronizedBuffer.java
2 // UnsynchronizedBuffer maintains the shared integer that is accessed by
3 // a producer thread and a consumer thread via methods set and get
4 public class UnsynchronizedBuffer implements Buffer {
5 {
6     private int buffer = -1; // shared by producer and consumer
7
8     // place value into buffer
9     public void set( int value ) throws InterruptedException
10    {
11        System.out.printf( "Producer writes\t%d", value );
12        buffer = value;
13    } // end method set
14
15    // return value from buffer
16    public int get() throws InterruptedException
17    {
18        System.out.printf( "Consumer reads\t%d", buffer );
19        return buffer;
20    } // end method get
21 } // end class UnsynchronizedBuffer

```

## Outline

Unsynchronized  
implementation of interface  
**Buffer**

**fixed  
BUFFER.java**



```

1 // Fig. 23.15: SharedBufferTest.java
2 // Application with 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.newCachedThreadPool();
12
13         // create UnsynchronizedBuffer to store ints
14         Buffer sharedLocation = new UnsynchronizedBuffer();
15
16         System.out.println(
17             "Action\t\tValue\tSum of Produced\tSum of Consumed");
18         System.out.println(
19             "-----\t-----\t-----\t-----\n");
20
21         // execute the Producer and Consumer, giving each of them access
22         // to sharedLocation
23         application.execute( new Producer( sharedLocation ) );
24         application.execute( new Consumer( sharedLocation ) );
25
26         application.shutdown(); // terminate application when tasks complete
27     } // end main
28 } // end class SharedBufferTest

```

## Outline

### SharedBufferTest.java

(1 of 3)

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



## Outline

### Shared Buffer Example.java

(2 of 3 )

Action	Value	Sum of Produced	Sum of 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	2	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	

— 1 is lost

— 2 is lost

— 5 is lost

— 6 is lost

— 7 read again

— 8 read again

— 9 is lost

Producer done producing

Terminating Producer

Consumer reads	10	47
Consumer reads	10	57
Consumer reads	10	67
Consumer reads	10	77

— 10 read again

— 10 read again

— 10 read again

Consumer read values totaling 77

Terminating Consumer

*(continued on next slide...)*



## Outline

### sharedbuffer OPTest.java

(3 of 3 )

Action	Value	Sum of Produced	Sum of Consumed
Consumer reads	-1		-1 —— reads -1 bad data
Producer writes	1	1	
Consumer reads	1		0
Consumer reads	1		1 —— I read again
Consumer reads	1		2 —— I read again
Consumer reads	1		3 —— I read again
Consumer reads	1		4 —— I read again
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	—— 5 is lost
Consumer reads	6		19

Consumer read values totaling 19

Terminating Consumer

Producer writes	7	28	—— 7 never read
Producer writes	8	36	—— 8 never read
Producer writes	9	45	—— 9 never read
Producer writes	10	55	—— 10 never read

Producer done producing

Terminating Producer



## 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



```

1 // Fig. 23.16: BlockingBuffer.java
2 // Creates a synchronized buffer using an ArrayBlockingQueue.
3 import java.util.concurrent.ArrayBlockingQueue;
4
5 public class BlockingBuffer implements Buffer {
6
7     private final ArrayBlockingQueue<Integer> buffer; // ←
8
9     public BlockingBuffer() {
10    buffer = new ArrayBlockingQueue<Integer>( 1 ); ←
11 } // end BlockingBuffer constructor
12
13 // place value into buffer
14 public void set( int value ) throws InterruptedException
15 {
16
17     buffer.put( value ); // place value in buffer
18     System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
19                      "Buffer cells occupied: ", buffer.size() );
20 } // end method set

```

## Outline

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

sync  
Creates a single element buffer of type **ArrayBlockingQueue**

BlockingBuffer.java



```
21  
22 // return value from buffer  
23 public int get() throws InterruptedException  
24 {  
25     int readValue = 0; // initialize value read from buffer  
26  
27     readValue = buffer.take(); // remove value from buffer  
28     System.out.printf( "%s %2d\t%s%d\n", "Consumer reads ",  
29         readValue, "Buffer cells occupied: ", buffer.size() );  
30  
31     return readValue;  
32 } // end method get  
33 } // end class BlockingBuffer
```

## Outline

### BlockingBu ffer.java

(2 of 2 )



```

1 // Fig. 23.17: BlockingBufferTest.java
2 // 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.newCachedThreadPool();
12
13         // create BlockingBuffer to store ints
14         Buffer sharedLocation = new BlockingBuffer();
15
16         application.execute( new Producer( sharedLocation ) );
17         application.execute( new Consumer( sharedLocation ) );
18
19         application.shutdown();
20     } // end main
21 } // end class BlockingBufferTest

```

## Outline

### BlockingBu ffer Test.java

(1 of 2 )

**Producer** and  
**Consumer** share  
the same synchronized  
**BUFFER**



## Outline

### **Blocking Queue Test.java**

(2 of 2 )

```
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



## 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** exception.



## 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.



```

1 // Fig. 23.18: SynchronizedBuffer.java
2 // Synchronizing access to shared data using Object
3 // methods wait and notify.
4 public class SynchronizedBuffer implements Buffer {
5
6     private int buffer = -1; // shared by producer and consumer
7     private boolean occupied = false; // whether the buffer is occupied
8
9     // place value into buffer
10    public synchronized void set( int value ) {
11
12        // while there are no empty locations, place the value
13        while ( occupied )
14        {
15            // output thread information and buffer information, then wait
16            System.out.println( "Producer tries to write." );
17            displayState( "Producer tries to write." );
18            wait(); // tell producer to wait
19        } // end while
20
21        buffer = value; // set new buffer value
22
23        // indicate producer cannot store another value
24        // until consumer retrieves current buffer value
25        occupied = true;
26
27        displayState( "Producer writes " + buffer );
28
29        notifyAll(); // tell consumer that a value is now available
30    } // end method set; releases monitor lock
  
```

## Outline

Synchronized implementation of interface **Buffer** that uses monitors and monitor locks

**producer**  
**Buffer.java**

of 3 )

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

enable state  
er



```

31
32 // return value from buffer
33 public synchronized int get() ← Synchronized method that
34 { allows a consumer
35 // while no data to read, place thr to read a value only if the
36 while ( !occupied ) buffer is full
37 {
38 // output thread information and buffer information, then wait
39 System.out.println( "Consumer tries to read." );
40 displayState( "E" );
41 wait(); ← consumer cannot
42 } // end while read so it must wait
43

```

## Outline

### Synchronous Buffer.java

(2 of 3 )



```

44 // indicate that producer can store another value
45 // because consumer just retrieved buffer value
46 occupied = false;
47
48 displayState( "Consumer reads " + buffer );
49
50 notifyAll(); // tell
51 Producer that a
52 return buffer; value is now available
53 } // end method get; releases lock on SynchronizedBuffer
54
55 // display current operation and buffer state
56 public void displayState( String operation )
57 {
58     System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
59     occupied );
60 } // end method displayState
61 } // end class SynchronizedBuffer

```

## Outline

### 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.



## Outline

### SharedBuffer Op Test2.java

(1 of 3 )

```

1 // Fig. 23.19: SharedBufferTest2.java
2 // 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 a newCachedThreadPool
11         ExecutorService application = Executors.newCachedThreadPool();
12
13         // create SyncronizedBuffer to store ints
14         Buffer sharedLocation = new SyncronizedBuffer();
15
16         System.out.printf( "%40s%<\t\t%<s\n%40s%<n\n", "Operation",
17             "Buffer", "Occupied", "-----", "-----\t\t-----" );
18
19         // execute the Producer and Consumer tasks
20         application.execute( new Producer( sharedLocation ) );
21         application.execute( new Consumer( sharedLocation ) );
22
23         application.shutdown();
24     } // end main
25 } // end class SharedBufferTest2

```

**Producer** and  
**Consumer** share  
the same synchronized  
**BUFFER**



## Outline

**sharedbuffer  
CP  
Test3.java**

(2 of 3 )

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...)*



*(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  
CP  
Test3.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..**



```

1 // Fig. 23.20: CircularBuffer.java
2 // Synchronizing access to a shared three-element bounded buffer.
3 public class CircularBuffer implements Buffer
4 {
5     private final int[] buffer = { -1, -1, -1 }; // shared buffer
6
7     private int occupiedCells = 0; // count number of buffers used
8     private int writeIndex = 0; // index of next element to write to
9     private int readIndex = 0; // index of next element to read
10
11    // place value into buffer
12    public synchronized void set( int value ) throws InterruptedException
13    {
14        // output thread information and buffer information, then wait;
15        // while no empty locations, place thread in block
16        while ( occupiedCells == buffer.length ) ← Determine whether buffer is
17        { full
18            System.out.printf( "Buffer is full. Producer waits.\n" );
19            wait(); // wait until a buffer cell is free
20        } // end while
21
22        buffer[ writeIndex ] = value; // set new buffer value
23
24        // update circular write index
25        writeIndex = ( writeIndex + 1 ) % buffer.length; ← Specify next write position
26
27        ++occupiedCells; // one more buffer cell is full
28        displayState( "Producer writes " + value );
29        notifyAll(); // notify threads waiting to read from buffer
30    } // end method set

```

## Outline

### CircularBuffer.java

(1 of 3 )



## Outline

### Circular Buffer Class

(2 of 3 )

```

31
32 // return value from buffer
33 public synchronized int get() throws InterruptedException
34 {
35     // wait until buffer has data, then read value;
36     // while no data to read, place thread
37     while ( occupiedCells == 0 ) ← Determine whether buffer is
38     {
39         System.out.printf( "Buffer is empty. Consumer waits.\n" );
40         wait(); // wait until a buffer cell is filled
41     } // end while
42
43     int readValue = buffer[ readIndex ]; // read value from buffer
44
45     // update circular read index
46     readIndex = ( readIndex + 1 ) % buffer.length; ← Specify the next read
47                                         location in the buffer
48     --occupiedCells; // one fewer buffer cells are occupied
49     displayState( "Consumer reads " + readValue );
50     notifyAll(); // notify threads waiting to write to buffer
51
52     return readValue;
53 } // end method get
54
55 // display current operation and buffer state
56 public void displayState( String operation )
57 {
58     // output operation and number of occupied buffer cells
59     System.out.printf( "%s% d)\n% s", operation,
60     " (buffer cells occupied: ", occupiedCells, " buffer cells: " );

```



```

61
62     for ( int value : buffer )
63         System.out.printf( " %2d ", value ); // output values in buffer
64
65     System.out.print( "\n" );
66
67     for ( int i = 0; i < buffer.length; i++ )
68         System.out.print( "----" );
69
70     System.out.print( "\n" );
71
72     for ( int i = 0; i < buffer.length; i++ )
73     {
74         if ( i == writeIndex && i == readIndex )
75             System.out.print( " WR" ); // both write and read index
76         else if ( i == writeIndex )
77             System.out.print( " W " ); // just write index
78         else if ( i == readIndex )
79             System.out.print( " R " ); // just read index
80         else
81             System.out.print( " " ); // neither index
82     } // end for
83
84     System.out.println( "\n" );
85 } // end method displayState
86 } // end class CircularBuffer

```

## Outline

### CircularBuffer for.java

(3 of 3 )



## Outline

### **CircularBuffer Test.java**

(1 of 5 )

```

1 // Fig. 23.21: CircularBufferTest.java
2 // Producer and Consumer 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.newCachedThreadPool();
12
13         // create CircularBuffer to store ints
14         CircularBuffer sharedLocation = new CircularBuffer();
15
16         // display the initial state of the CircularBuffer
17         sharedLocation.displayState( "Initial State" );
18
19         // execute the Producer and Consumer tasks
20         application.execute( new Producer( sharedLocation ) );
21         application.execute( new Consumer( sharedLocation ) );
22
23         application.shutdown();
24     } // end main
25 } //end class CircularBuffer Test

```

**Producer** and  
**Consumer** share  
the same synchronized  
circular **Buffer**



## Outline

### CIRCULAR BUFFER TEST.java

(2 of 5 )

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... )*



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...)*

## Outline

### **CircularBuffer for Test.java**

(3 of 5 )



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... )*

## Outline

### **CIRCULAR BUFFER TEST.java**

(4 of 5 )



## Outline

### **CIRCULARBUFFER FOR TEST.JAVA**

(5 of 5 )

Consumer reads 8 (buffer cells occupied: 1)

buffer cells: 7 8 9



Consumer reads 9 (buffer cells occupied: 0)

buffer cells: 7 8 9



Producer writes 10 (buffer cells occupied: 1)

buffer cells: 10 8 9



Producer done producing

Terminating Producer

Consumer reads 10 (buffer cells occupied: 0)

buffer cells: 10 8 9



Consumer read values totaling: 55

Terminating Consumer



# 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



# 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**



# 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 **illegal monitor state exception** 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.



```

1 // Fig. 23.22: SynchronizedBuffer.java
2 // Synchronizing access to a shared integer using the Lock and Condition
3 // interfaces
4 import java.util.concurrent.locks.Lock;
5 import java.util.concurrent.locks.ReentrantLock;
6 import java.util.concurrent.locks.Condition;
7
8 public class SynchronizedBuffer implements Buffer {
9
10    // Lock to control synchronization with this buffer
11    private final Lock accessLock = new ReentrantLock();
12
13    // conditions to control reading and writing
14    private final Condition canWrite = accessLock.newCondition();
15    private final Condition canRead = accessLock.newCondition();
16
17    private int buffer = -1; // shared by producer and consumer threads
18    private boolean occupied = false; // whether buffer is occupied
19
20    // place int value into buffer
21    public void set( int value ) throws InterruptedException
22    {
23        accessLock.lock(); // lock this object
24
25        // output thread information and buffer information
26        try
27        {

```

## Outline

Synchronized implementation of interface **Buffer** that uses **Locks** and **Conditions** (1 of 4)

**Condition**  
**Condition**  
indicating when a consumer can read

Manually acquire the lock to implement mutual exclusion



## Outline

```

28 // while buffer is not empty, place thread in waiting state
29 while ( occupied )
30 {
31     System.out.println( "Producer tries to write." );
32     displayState( "Buffer full. Producer waits." );
33     canWrite.await(); // wait until buffer is empty ←
34 } // end while
35
36 buffer = value; // set new buffer value
37
38 // indicate producer cannot store another value
39 // until consumer retrieves current buffer value
40 occupied = true;
41
42 displayState( "Producer writes " + buffer );
43
44 // signal thread waiting
45 canRead.signal(); ←
46 } // end try
47 finally
48 {
49     accessLock.unlock(); ← unlock
50 } // end finally
51 } // end method set
52

```

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

### Synchronous Buffer.java

(3 of 4 )

```

53 // return value from buffer
54 public int get() throws InterruptedException
55 {
56     int readValue = 0; // initialize
57     accessLock.lock(); // lock
58
59     // output thread information
60     try
61     {
62         // while no data to read, place thread in waiting state
63         while ( !occupied )
64         {
65             System.out.println( "Consumer tries to read." );
66             displayState( "Buffer empty. Consumer waits." );
67             canRead.await(); // wait until buffer is full
68         } // end while
69
70         // indicate that producer can store another value
71         // because consumer just retrieved buffer value
72         occupied = false;
73
74         readValue = buffer; // retrieve value from buffer
75         displayState( "Consumer reads " + readValue );
76
77         // signal thread waiting for write
78         canWrite.signal();
79     } // 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



```
80     finally
81     {
82         accessLock.unlock(); // unlock
83     } // end finally
84
85     return readValue;
86 } // end method get
87
88 // display current operation and buffer state
89 public void displayState( String operation )
90 {
91     System.out.printf( "%-40s%t\t%b\n\n", operation, buffer,
92     occupied );
93 } // end method displayState
94 } // end class SynchronizedBuffer
```

Release the lock so  
producer can write

## Outline

### Synchronized Buffer.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.



```

1 // Fig. 23.23: SharedBufferTest2.java
2 // 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.newCachedThreadPool();
12
13         // create SynchronizedBuffer to store ints
14         Buffer sharedLocation = new SynchronizedBuffer();
15
16         System.out.printf( "%-40s%<\t\t%<s\n%-40s%<s\n", "Operation",
17             "Buffer", "Occupied", "-----", "-----\t\t-----" );
18
19         // execute the Producer and Consumer tasks
20         application.execute( new Producer( sharedLocation ) );
21         application.execute( new Consumer( sharedLocation ) );
22
23         application.shutdown();
24     } // end main
25 } // 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...)

## Outline

### SHARED BUFFER OP TEST2.java

(1 of 3 )



*(continued from previous slide...)*

## Outline

**sharedbuffer  
op  
TestMain.java**

(2 of 3 )

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  
Op  
TestMain.java**

(3 of 3 )

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 **java.awt.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
dol nBackgr ound	Defines a long computation and is called in a worker thread.
done	Executes on the event dispatch thread when dol nBackgr ound returns.
execut e	Schedules the Swi ngWor ker 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 dol nBackgr ound).
publ i sh	Sends intermediate results from the dol nBackgr ound method to the process method for processing on the event dispatch thread.
pr ocess	Receives intermediate results from the publ i sh method and processes these results on the event dispatch thread.
set Pr ogress	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



## Outline

```

1 // Fig. 23.25: BackgroundCalculator.java
2 // SwingWorker subclass for calculating Fibonacci numbers
3 // in a background thread.
4 import javax.swing.SwingWorker;
5 import javax.swing.JLabel;
6 import java.util.concurrent.ExecutionException;
7
8 public class BackgroundCalculator extends SwingWorker<String, Object> {
9
10    private final int n; // Fibonacci number to calculate
11    private final JLabel resultJLabel; // JLabel to display the result
12
13    // constructor
14    public BackgroundCalculator( int number, JLabel label )
15    {
16        n = number;
17        resultJLabel = label;
18    } // end BackgroundCalculator constructor
19
20    // Long-running code to be run in a worker thread
21    public String doInBackground()
22    {
23        long nthFib = fibonacci( n );
24        return String.valueOf( nthFib );
25    } // end method doInBackground
26

```

Create a subclass of  
SwingWorker to

(1 of 2 )

Possibly lengthy Fibonacci  
calculation to perform in  
the background



```

27 // code to run on the event dispatch thread when doInBackground returns
28 protected void done()
29 {
30     try
31     {
32         // get the result of doInBackground and
33         resultJLabel.setText( get() ); ←
34     } // end try
35     catch ( InterruptedException ex )
36     {
37         resultJLabel.setText( "Interrupted while waiting for results." );
38     } // end catch
39     catch ( ExecutionException ex )
40     {
41         resultJLabel.setText(
42             "Error encountered while performing calculation." );
43     } // end catch
44 } // end method done
45
46 // recursive method fibonacci; calculates nth Fibonacci number
47 public long fibonacci( long number )
48 {
49     if ( number == 0 || number == 1 )
50         return number;
51     else
52         return fibonacci( number - 1 ) + fibonacci( number - 2 );
53 } // end method fibonacci
54 } // end class BackgroundCalculator

```

Display the calculation  
results when done

## Outline

### BackgroundCalculator.java

(2 of 2)



## 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.

---



```

1 // Fig. 23.26: FibonacciNumbers.java
2 // Using SwingWorker to perform a long calculation with
3 // intermediate results displayed in a GUI.
4 import java.awt.GridLayout;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.ActionListener;
7 import javax.swing.JButton;
8 import javax.swing.JFrame;
9 import javax.swing.JPanel;
10 import javax.swing.JLabel;
11 import javax.swing.JTextField;
12 import javax.swing.border.TitledBorder;
13 import javax.swing.border.LineBorder;
14 import java.awt.Color;
15 import java.util.concurrent.ExecutionException;
16
17 public class FibonacciNumbers extends JFrame
18 {
19     // components for calculating the Fibonacci of a user-entered number
20     private final JPanel worker JPanel =
21         new JPanel( new GridLayout( 2, 2, 5, 5 ) );
22     private final JTextField number JTextField = new JTextField();
23     private final JButton go JButton = new JButton( "Go" );
24     private final JLabel fibonacci JLabel = new JLabel();
25
26     // components and variables for getting the next Fibonacci number
27     private final JPanel event Thread JPanel =
28         new JPanel( new GridLayout( 2, 2, 5, 5 ) );
29     private int n1 = 0; // initialize with first Fibonacci number

```

## Outline

### FibonacciNumbers.java

(1 of 5 )



## Outline

### Fibonacci numbers.java

(2 of 5 )

```
30 private int n2 = 1; // initialize with second Fibonacci number
31 private int count = 1;
32 private final JLabel nJLabel = new JLabel( "Fibonacci of 1: " );
33 private final JLabel nFibonacci JLabel =
34     new JLabel( String.valueOf( n2 ) );
35 private final JButton nextNumber JButton = new JButton( "Next Number" );
36
37 // constructor
38 public FibonacciNumbers()
39 {
40     super( "Fibonacci Numbers" );
41     setLayout( new GridLayout( 2, 1, 10, 10 ) );
42
43     // add GUI components to the SwingWorker panel
44     worker JPanel .setBorder( new TitleBorder(
45         new LineBorder( Color.BLACK ), "With SwingWorker" ) );
46     worker JPanel .add( new JLabel( "Get Fibonacci of:" ) );
47     worker JPanel .add( number JTextField );
48     goJButton.addActionListener(
49         new ActionListener()
50     {
51         public void actionPerformed( ActionEvent event )
52         {
53             int n;
```



## Outline

### **Fibonacci numbers.java**

(3 of 5 )

```
55     try
56     {
57         // retrieve user's input as an integer
58         n = Integer.parseInt( number JTextField.getText() );
59     } // end try
60     catch( NumberFormatException ex )
61     {
62         // display an error message if the user did not
63         // enter an integer
64         fibonacci JLabel.setText( "Enter an integer." );
65         return;
66     } // end catch
67
68     // indicate that the calculation has begun
69     fibonacci JLabel.setText( "Calculating..." );
70
71     // create a task to perform calculation in background
72     BackgroundCalculator task =
73         new BackgroundCalculator( n, fibonacci JLabel );
74     task.execute(); // execute the task
75 } // end method actionPerformed
76 } // end anonymous inner class
77 ); // end call to addActionListener
78 worker JPanel.add( go JButton );
79 worker JPanel.add( fibonacci JLabel );
80
```



```

1 // add GUI components to the event-dispatching thread panel
2 eventThreadJPanel . setBorder( new TitledBorder(
3     new LineBorder( Color . BLACK ), "Without SwingWorker" ) );
4 eventThreadJPanel . add( nJLabel );
5 eventThreadJPanel . add( nFibonacciJLabel );
6 nextNumberJButton . addActionListener(
7     new ActionListener()
8     {
9         public void actionPerformed( ActionEvent event )
10        {
11            // calculate the Fibonacci number after n2
12            int temp = n1 + n2;
13            n1 = n2;
14            n2 = temp;
15            ++count;
16
17            // display the next Fibonacci number
18            nJLabel . setText( "Fibonacci of " + count + ":" );
19            nFibonacciJLabel . setText( String . valueOf( n2 ) );
20        } // end method actionPerformed
21    } // end anonymous inner class
22); // end call to addActionListener
23 eventThreadJPanel . add( nextNumberJButton );
24
25 add( workerJPanel );
26 add( eventThreadJPanel );
27 setSize( 275, 200 );
28 setVisible( true );
29 } // end constructor
30

```

## Outline

### Fibonacci.java numbers.java

(4 of 5)



```

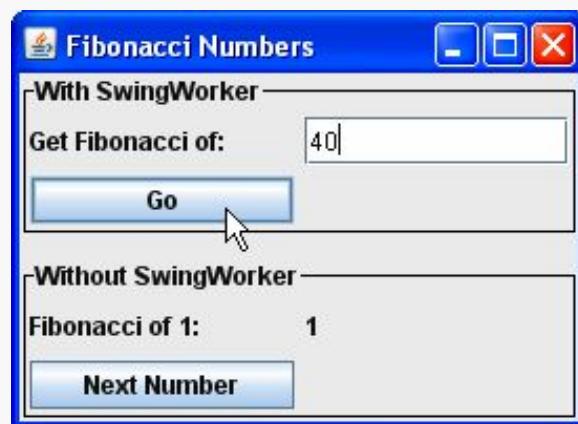
111 // main method begins program execution
112 public static void main( String[] args )
113 {
114     FibonacciNumbers application = new FibonacciNumbers();
115     application.setDefaultCloseOperation( EXIT_ON_CLOSE );
116 } // end main
117} // end class FibonacciNumbers

```

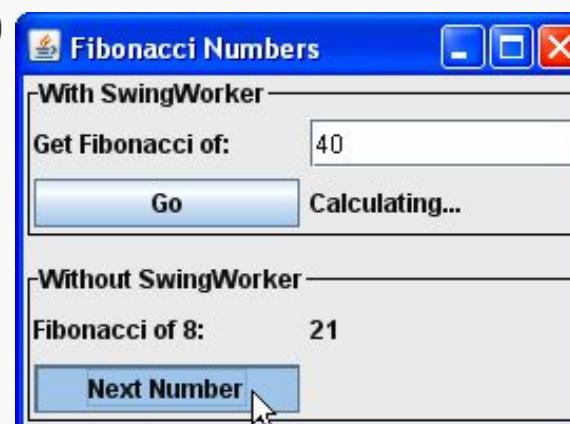
## Outline

### FibonacciNumbers.java

a)

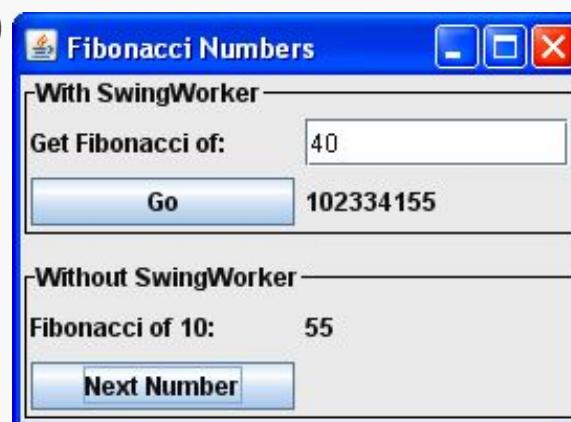


b)



(5 of 5 )

c)



## 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



```

1 // Fig. 23.27: PrimeCalculator.java
2 // Calculates the first n primes, displaying them as they are found.
3 import javax.swing.JTextArea;
4 import javax.swing.JLabel;
5 import javax.swing.JButton;
6 import javax.swing.SwingWorker;
7 import java.util.Random;
8 import java.util.List;
9 import java.util.concurrent.ExecutionException;
10
11 public class PrimeCalculator extends SwingWorker<Integer, Integer>
12 {
13     private final Random generator = new Random();
14     private final JTextArea intermediateJTextArea; // displays found primes
15     private final JButton getPrimesJButton;
16     private final JButton cancelJButton;
17     private final JLabel statusJLabel; // displays status of calculation
18     private final boolean primes[]; // boolean array for finding primes
19     private boolean stopped = false; // flag indicating cancellation
20
21     // constructor
22     public PrimeCalculator( int max, JTextArea intermediate, JLabel status,
23                           JButton getPrimes, JButton cancel )
24     {
25         intermediateJTextArea = intermediate;
26         statusJLabel = status;
27         getPrimesJButton = getPrimes;
28         cancelJButton = cancel;
29         primes = new boolean[ max ];
30

```

## Outline

### **Primecalculator.java**

(1 of 5 )



## Outline

### PrimeCalculator Worker.java

(2 of 5 )

```

31 // initialize all primes array values to true
32     for ( int i = 0; i < max; i ++ )
33         primes[ i ] = true;
34 } // end constructor
35
36 // finds all primes up to max using the Sieve of Eratosthenes
37 public Integer doInBackground()
38 {
39     int count = 0; // the number of primes found
40
41     // starting at the third value, cycle through the array and put
42     // false as the value of any greater number that is a multiple
43     for ( int i = 2; i < primes.length; i ++ )
44     {
45         if ( stopped ) // if calculation has been canceled
46             return count;
47         else
48         {
49             setProgress( 100 * ( i + 1 ) / primes.length ); ←
50
51             try
52             {
53                 Thread.currentThread().sleep( generator.nextInt( 5 ) );
54             } // end try
55             catch ( InterruptedException ex )
56             {
57                 statusJLabel.setText( "Worker thread interrupted" );
58                 return count;
59             } // end catch
60         }
61     }
62 }
```

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



```

61     if ( primes[ i ] ) // i is prime
62     {
63         publish( i ); // make
64         ++count;
65
66         for ( int j = i + i; j < primes.length; j += i )
67             primes[ j ] = false; // i is not prime
68     } // end if
69 } // end else
70 } // end for
71
72 return count;
73 } // end method doInBackground
74
75 // displays published values in primes list
76 protected void process( List< Integer > publishedVals ) ←
77 {
78     for ( int i = 0; i < publishedVals.size(); i++ )
79         intermediateTextArea.append( publishedVals.get( i ) + "\n" );
80 } // end method process

```

Publish each prime as it is discovered

prime list

## Outline

### PrimeCalculator User.java

(3 of 5)

Process all the published prime values



## Outline

### **PrimeCalculation User.java**

(4 of 5 )

```
81
82 // code to execute when doInBackground completes
83 protected void done()
84 {
85     getPrimesJButton.setEnabled( true ); // enable Get Primes button
86     cancelJButton.setEnabled( false ); // disable Cancel button
87
88     int numPrimes;
89
90     try
91     {
92         numPrimes = get(); // retrieve doInBackground return value
93     } // end try
94     catch ( InterruptedException ex )
95     {
96         statusJLabel.setText( "Interrupted while waiting for results." );
97         return;
98     } // end catch
99     catch ( ExecutionException ex )
100    {
101        statusJLabel.setText( "Error performing computation." );
102        return;
103    } // end catch
```



```
104     statusJLabel.setText( "Found " + numPrimes + " primes." );
105 } // end method done
106
107
108 // sets flag to stop looking for primes
109 public void stopCalculation()
110 {
111     stopped = true;
112 } // end method stopCalculation
113} // end class PrimeCalculator
```

## Outline

### Primecalculator User.java

(5 of 5 )



```
1 // Fig. 23.28: FindPrimes.java
2 // Using a SwingWorker to display prime numbers and update a JProgressBar
3 // while the prime numbers are being calculated.
4 import javax.swing.JFrame;
5 import javax.swing.JTextField;
6 import javax.swing.JTextArea;
7 import javax.swing.JButton;
8 import javax.swing.JProgressBar;
9 import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.ScrollPaneConstants;
13 import java.awt.BorderLayout;
14 import java.awt.GridLayout;
15 import java.awt.event.ActionListener;
16 import java.awt.event.ActionEvent;
17 import java.util.concurrent.ExecutionException;
18 import java.beans.PropertyChangeListener;
19 import java.beans.PropertyChangeEvent;
20
21 public class FindPrimes extends JFrame
22 {
23     private final JTextField highestPrimeTextField = new JTextField();
24     private final JButton getPrimes JButton = new JButton("Get Primes");
25     private final JTextArea displayPrimesJTextArea = new JTextArea();
26     private final JButton cancel JButton = new JButton("Cancel");
27     private final JProgressBar progressBar = new JProgressBar();
28     private final JLabel statusJLabel = new JLabel();
29     private PrimeCalculator calculator;
30 }
```

## Outline

### FindPrimes.java

(1 of 6)



```

31 // constructor
32 public FindPrimes()
33 {
34     super( "Finding Primes with SwingWorker" );
35     setLayout( new BorderLayout() );
36
37     // initialize panel to get a number from the user
38     JPanel northJPanel = new JPanel();
39     northJPanel.add( new JLabel( "Find primes less than: " ) );
40     highestPrime JTextField.setColumns( 5 );
41     northJPanel.add( highestPrime JTextField );
42     getPrimes JButton.addActionListener(
43         new ActionListener()
44     {
45         public void actionPerformed( ActionEvent e )
46     {
47             progressBar.setValue( 0 ); // reset JProgressBar
48             displayPrimes JTextArea.setText( "" ); // clear JTextArea
49             statusJLabel.setText( "" ); // clear JLabel
50
51             int number;
52
53             try
54             {
55                 // get user input
56                 number = Integer.parseInt(
57                     highestPrime JTextField.getText() );
58             } // end try

```

## Outline

### **FindPrimes.java**

(2 of 6)



## Outline

### FindPrimes.java

(3 of 6)

```

59     catch ( NumberFormatException ex )
60     {
61         statusJLabel.setText( "Enter an integer." );
62         return;
63     } // end catch
64
65     // construct a new PrimeCalculator object
66     calculator = new PrimeCalculator( number,
67                                     displayPrimesJTextArea, statusJLabel, getPrimesJButton,
68                                     cancelButton );
69
70     // listen for progress bar property changes
71     calculator.addPropertyChangeListener(
72         new PropertyChangeListener()
73     {
74         public void propertyChange( PropertyChangeEvent e )
75         {
76             // if the changed property is progress,
77             // update the progress bar
78             if ( e.getPropertyName().equals( "progress" ) )
79             {
80                 int newValue = ( Integer ) e.getNewValue();
81                 progressJProgressBar.setValue( newValue );
82             } // end if
83         } // end method propertyChange
84     } // end anonymous inner class
85 ); // end call to addPropertyChangeListener

```



## Outline

### FindPrimes.java

(4 of 6 )

```
86
87      // disable Get Primes button and enable Cancel button
88      getPrimesJButton.setEnabled( false );
89      cancelJButton.setEnabled( true );
90
91      calculator.execute(); // execute the PrimeCalculator object
92  } // end method actionPerformed
93 } // end anonymous inner class
94 ); // end call to addActionListener
95 northJPanel.add( getPrimesJButton );
96
97 // add a scrollable JList to display results of calculation
98 displayPrimesTextArea.setEditable( false );
99 add( new JScrollPane( displayPrimesTextArea,
100     ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
101     ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER ) );
102
103 // initialize a panel to display cancel JButton,
104 // progressJProgressBar, and statusJLabel
105 JPanel southJPanel = new JPanel( new GridLayout( 1, 3, 10, 10 ) );
106 cancelJButton.setEnabled( false );
107 cancelJButton.addActionListener(
108     new ActionListener()
109     {
110         public void actionPerformed( ActionEvent e )
111         {
112             calculator.stopCalculation(); // cancel the calculation
113         } // end method actionPerformed
114     } // end anonymous inner class
115 ); // end call to addActionListener
```

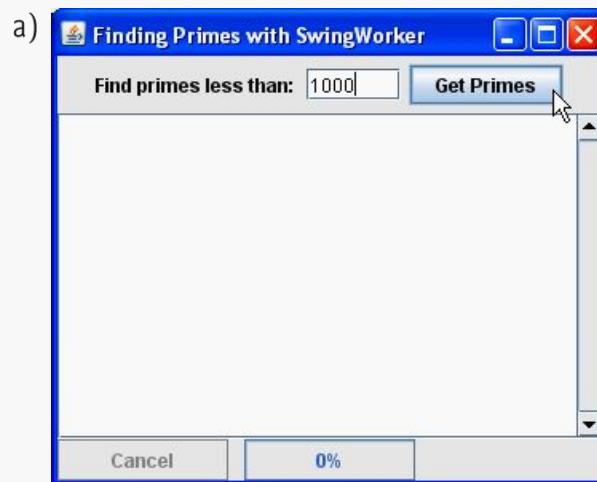


## Outline

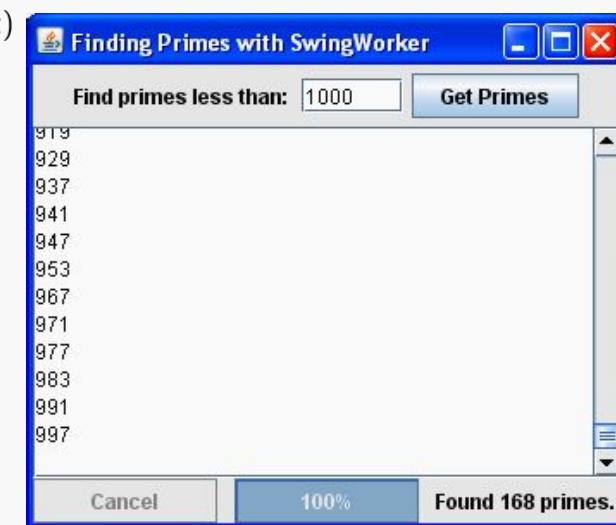
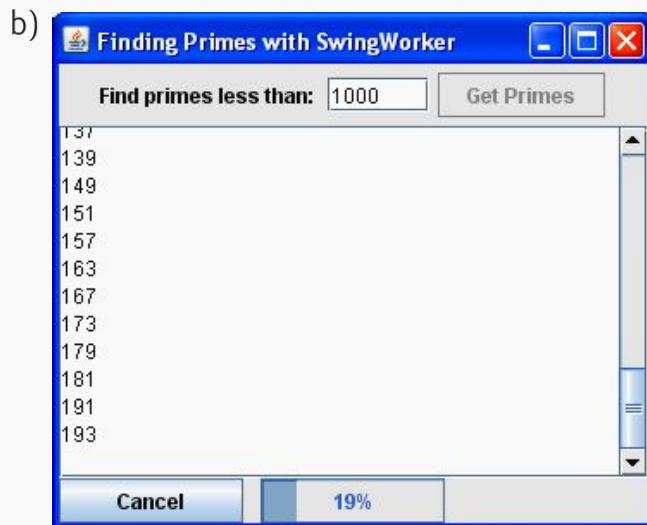
### **FindPrimes.java**

(5 of 6 )

```
116     southJPanel . add( cancel JButton );
117     progressJProgressBar . setStringPainted( true );
118     southJPanel . add( progressJProgressBar );
119     southJPanel . add( statusJLabel );
120
121     add( northJPanel , BorderLayout . NORTH );
122     add( southJPanel , BorderLayout . SOUTH );
123     setSize( 350 , 300 );
124     setVisible( true );
125 } // end constructor
126
127 // main method begins program execution
128 public static void main( String[] args )
129 {
130     FindPrimes application = new FindPrimes();
131     application.setDefaultCloseOperation( EXIT_ON_CLOSE );
132 } // end main
133} // end class FindPrimes
```



## Outline



(6 of 6 )

Finding Primes  
Java



# 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

