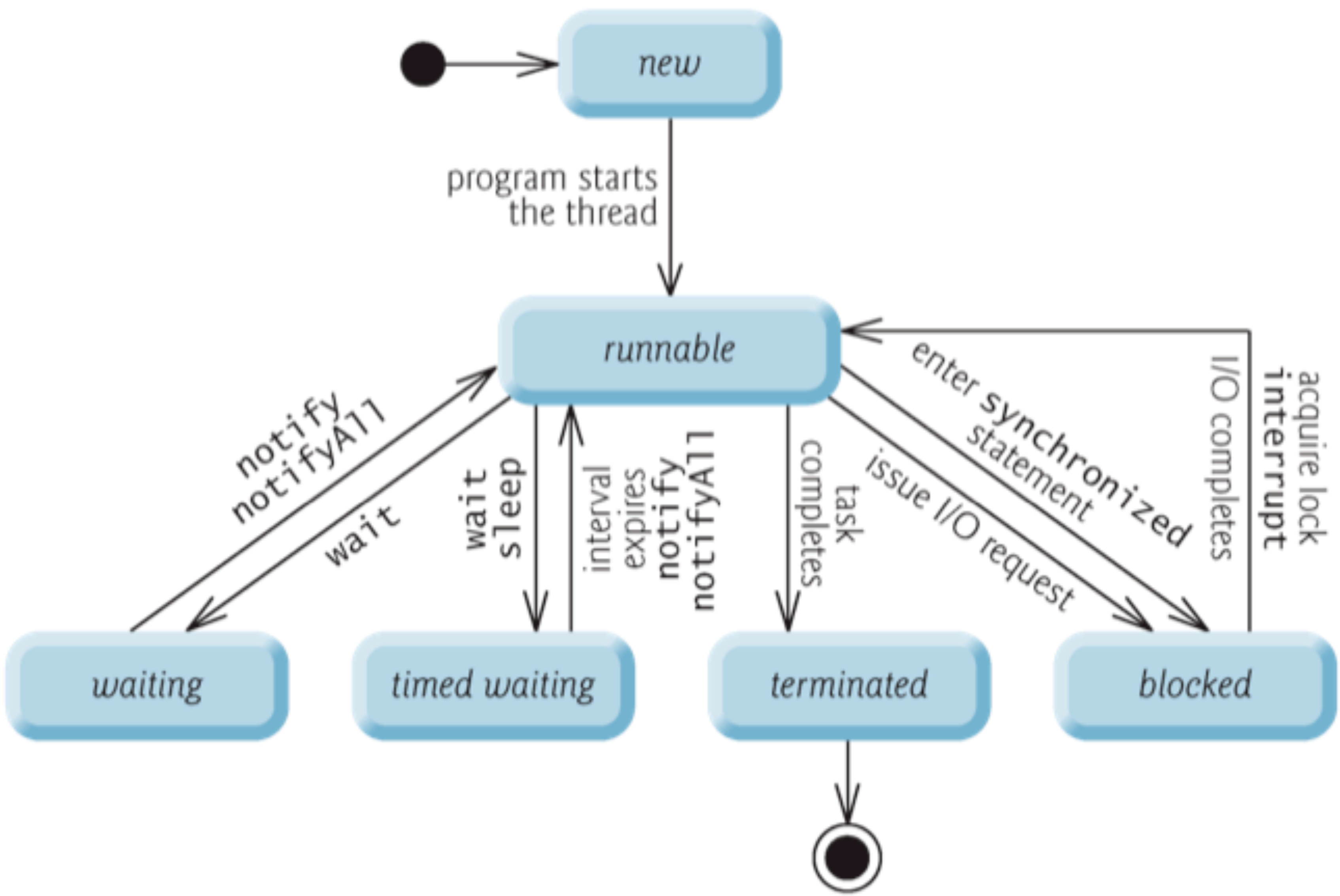
Multithreading

Introduction

- Java makes concurrency available to you through the language and APIs.
- You specify that an application contains separate threads of execution
  - each thread has its own **method-call stack** and program counter
  - can execute concurrently with other threads while sharing applicationwide resources such as memory with them.
- This capability is called multithreading.

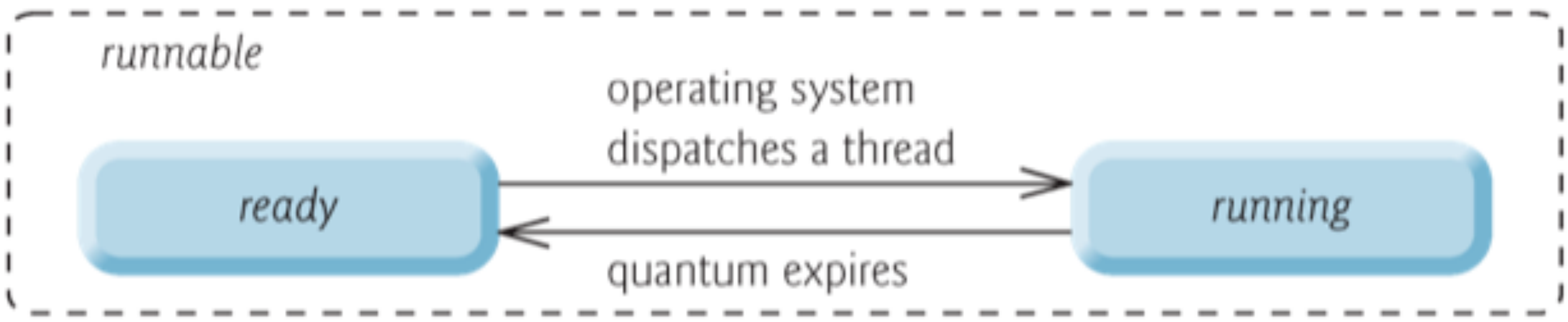
Thread States: Life Cycle of a Thread

- At any time, a thread is said to be in one of several thread states
- A new thread begins its life cycle in the new **state.**
- Remains there until started, which places it in the runnable **state—considered to be executing its task.**
- A *runnable* thread can transition 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 it to continue executing.


new
program starts
the thread
runnable
notify
notifyAll
wait
wait
sleep
interval
expires
notify
notifyAll
task
completes
issue I/O request
enter synchronized
statement
I/O completes
acquire lock
interrupt
waiting
timed waiting
terminated
blocked

Thread States: Life Cycle of a Thread (cont.)

- At the operating-system level, Java's *runnable* state typically encompasses two separate states



Thread States: Life Cycle of a Thread (cont.)

- Typically, each thread is given a a quantum or timeslice in which to perform its task.
- The process that an operating system uses to determine which thread to dispatch is called thread scheduling.

Thread States: Lifecycle of a Thread (cont.)

- Thread priority: helps determine the order in which threads are scheduled.
- Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads.
- Thread priorities cannot guarantee the order in which threads execute.
- Do not explicitly create and use Thread objects to implement concurrency.
- Rather, use the `Executor` interface.

Thread States: Lifecycle of a Thread (cont.)

- Most operating systems support timeslicing, which enables threads of equal priority to share a processor.
- An operating system's thread scheduler determines which thread runs next.
- One simple thread-scheduler implementation keeps the highest-priority thread running at all times and, if there's more than one highest-priority thread, ensures that all such threads execute for a quantum each in round-robin fashion. This process continues until all threads run to completion.

Thread States: Lifecycle of a Thread (cont.)

- When a higher-priority thread enters the *ready* state, the operating system generally preempts the currently *running* thread (an operation known as preemptive scheduling).
- Higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads.
- Indefinite postponement is sometimes referred to as starvation.
- Operating systems employ a technique called *aging* to prevent starvation.

Thread States: Lifecycle of a Thread (cont.)

- Java provides higher-level concurrency utilities to hide much of this complexity and make multithreaded programming less error prone.
- Thread priorities are used behind the scenes to interact with the operating system, but most programmers who use Java multithreading will not be concerned with setting and adjusting thread priorities.

Creating and Executing Threads in Java

- A Runnable object represents a "task" that can execute concurrently with other tasks.
- The `Runnable` interface declares the single method run, which contains the code that defines the task that a `Runnable` object should perform.
- When a thread executing a `Runnable` is created and started, the thread calls the `Runnable` object's `run` method, which executes in the new thread.

```java
public class Worker implements Runnable
{
    public static void main (String[] args)
    {
        System.out.println("This is currently running on the main thread, " +
            "the id is: " + Thread.currentThread().getId());
        Worker worker = new Worker();
        Thread thread = new Thread(worker);
        thread.start();
    }

    @Override
    public void run()
    {
        System.out.println("This is currently running on a separate thread, " +
                "the id is: " + Thread.currentThread().getId());

    }
}
```

When this code is run, here’s the output that I got:

- This is currently running on the main thread, the id is: 1
- This is currently running on a separate thread, the id is: 9

Creating and Executing Threads with `Executor` Framework

Creating and Executing Threads with `Executor` Framework (cont.)

- Recommended that you use the Executor interface to manage the execution of `Runnable` objects for you.
  - Typically creates and manages a group of threads called a thread pool to execute `Runnable`s.
- `Executor`s can reuse existing threads and can improve performance by optimizing the number of threads.
- `Executor` method execute accepts a `Runnable` as an argument.
- An `Executor` assigns every `Runnable` to one of the available threads in the thread pool.
- If there are no available threads, the `Executor` creates a new thread or waits for a thread to become available.

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

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

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

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

```

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

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

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

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

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

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

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

```

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

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

```
Starting Executor
Tasks started, main ends

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

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

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

Thread Synchronization

- When multiple threads share an object and it is modified by one or more of them, indeterminate results may occur, unless access to the shared object is managed properly.

- The problem can be solved by giving only one thread at a time *exclusive* access to code that manipulates the shared object.
  - During that time, other threads desiring to manipulate the object are kept waiting.
  - When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting is allowed to proceed.

Thread Synchronization (cont.)

- This process, called thread synchronization, coordinates access to shared data by multiple concurrent threads.
  - Ensures that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called mutual exclusion.

Thread Synchronization (cont.)

- A common way to perform synchronization is to use Java's built-in monitors.
  - Every object has a monitor and a monitor lock (or intrinsic lock).
  - Can be held by a maximum of only one thread at any time.
  - A thread must acquire the lock before proceeding with the operation.
  - Other threads attempting to perform an operation that requires the same lock will be *blocked*.
- To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a synchronized statement.
  - Said to be guarded by the monitor lock

Thread Synchronization (cont.)

- The `synchronized` statements are declared using the synchronized keyword:
  - ```
    synchronized ( object )
    {
        statements
    } // end synchronized statement
    ```
- where *object* is the object whose monitor lock will be acquired
  - *object* is normally `this` if it's the object in which the `synchronized` statement appears.
- When a `synchronized` statement finishes executing, the object's monitor lock is released.
- Java also allows synchronized methods.
    ```

Unsynchronized Data Sharing

- A `SimpleArray` object in the example will be shared across multiple threads.
- Will enable those threads to place `int` values into `array`.
- Line 26 puts the thread that invokes `add` to sleep for a random interval from 0 to 499 milliseconds.
  - This is done to make the problems associated with unsynchronized access to shared data more obvious.

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

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

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

```

```java
   // 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

```

```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()
     {
        return "\nContents of SimpleArray:\n" + Arrays.toString( array );
     } // end method toString
  } // end class SimpleArray
```

Unsynchronized Data Sharing (cont.)

- Class `ArrayWriter` implements the interface `Runnable` to define a task for inserting values in a `SimpleArray` object.
- The task completes after three consecutive integers beginning with `startValue` are added to the `SimpleArray` object.

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

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

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

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

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

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

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

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

      executor.shutdown();

```

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

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

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

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

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

Synchronized Data Sharing—Making Operations Atomic

- The output errors can be attributed to the fact that the shared object, `SimpleArray`, is not thread safe.
- If one thread obtains the value of `writeIndex`, there is no guarantee that another thread cannot come along and increment `writeIndex` before the first thread has had a chance to place a value in the array.
- If this happens, the first thread will be writing to the array based on a stale value of `writeIndex`—a value that is no longer valid.

Synchronized Data Sharing—Making Operations Atomic (cont.)

- An atomic operation cannot be divided into smaller suboperations.
- Can simulate atomicity by ensuring that only one thread carries out the three operations at a time.
- Atomicity can be achieved using the `synchronized` keyword.

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

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

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

```

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

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

```

```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()
   {
      return "\nContents of SimpleArray:\n" + Arrays.toString( array );
   } // end method toString
} // end class SimpleArray
```

```
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
```