Lecture 10: Multithreading and Parallel Programming (Ch 32)

Adapted by Fangzhen Lin for COMP3021 from Y. Danial Liang's PowerPoints for Introduction to Java Programming, Comprehensive Version, 9/E, Pearson, 2013.

Objectives

☞ To get an overview of multithreading (§32.2).

☞ To develop task classes by implementing the <u>Runnable</u> interface (§32.3).

☞ To create threads to run tasks using the <u>Thread</u> class (§32.3).

☞ To control threads using the methods in the <u>Thread</u> class (§32.4).

☞ To control animations using threads (§32.5, §32.7).

☞ To run code in the event dispatch thread (§32.6).

☞ To execute tasks in a thread pool (§32.8).

☞ To use synchronized methods or blocks to synchronize threads to avoid race conditions (§32.9).

☞ To synchronize threads using locks (§32.10).

☞ To facilitate thread communications using conditions on locks (§§32.11-32.12).

☞ To use blocking queues to synchronize access to an array queue, linked queue, and priority queue (§32.13).

☞ To restrict the number of accesses to a shared resource using semaphores (§32.14).

☞ To use the resource-ordering technique to avoid deadlocks (§32.15).

☞ To describe the life cycle of a thread (§32.16).

☞ To create synchronized collections using the static methods in the <u>Collections</u> class (§32.17).

☞ To develop parallel programs using the Fork/Join Framework (§32.18).

☞ To run time-consuming tasks in a <u>SwingWorker</u> rather than in the event dispatch thread (§32.19).

☞ To display the completion status of a task using <u>JProgressBar</u> (§32.20).

2

Threads Concept

Multiple threads on multiple CPUs

Thread 1	
Thread 2	
Thread 3	
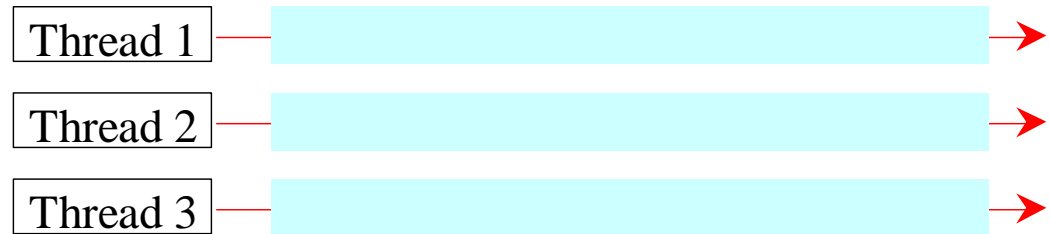
Multiple threads sharing a single CPU

Thread 1	
Thread 2	
Thread 3	
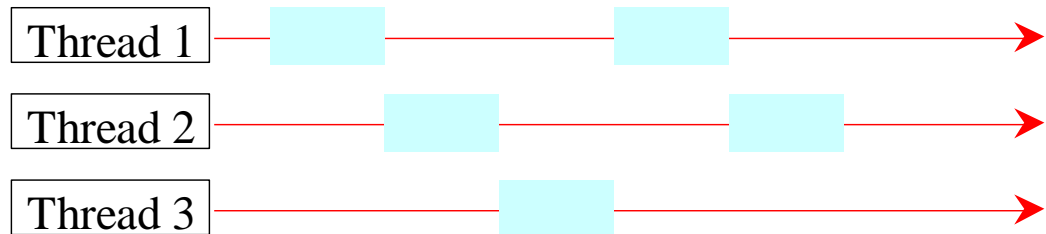
Creating Tasks and Threads

| java.lang.Runnable | ◁- - - - - | TaskClass |

```java
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```java
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Example:
Using the `Runnable` Interface to Create and Launch Threads

☞ Objective: Create and run three threads:

- The first thread prints the letter *a* 100 times.
- The second thread prints the letter *b* 100 times.
- The third thread prints the integers 1 through 100.

TaskThreadDemo

run()

☞ The run() methods in a task class specifies how to perform the task. It's automatically invoked by JVM when a thread is started.

☞ You should not invoke it: doing so merely executes this method in the same thread; no new thread is started.

The Thread Class

«interface» *java.lang.Runnable*	

java.lang.Thread	
+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

The Static yield() Method

You can use the yield() method to temporarily release time for other threads. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:

```
public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        Thread.yield();
    }
}
```

Every time a number is printed, the print100 thread is yielded. So, the numbers are printed after the characters.

The Static sleep(milliseconds) Method

The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:

```java
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```
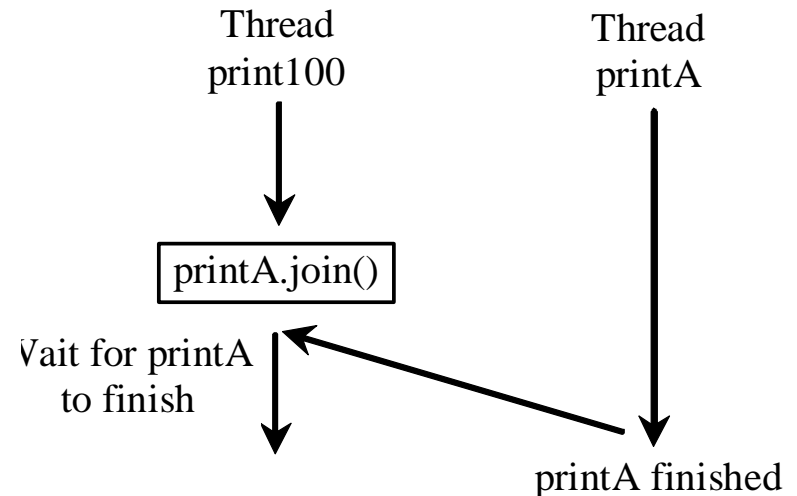
Every time a number (>= 50) is printed, the print100 thread is put to sleep for 1 millisecond.

- The sleep() method may throw an InterruptedException, which is a checked exception.

- This rarely occurs, but you have to catch it as it is a checked exception.

- The same hold for the join() method.

The join() Method

You can use the join() method to force one thread to wait for another thread to finish. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:

```java
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
`
```

Thread
print100

Thread
printA

printA.join()

Wait for printA
to finish

printA finished

The numbers after 50 are printed after thread printA is finished.

isAlive(), interrupt(), and isInterrupted()

The isAlive() method is used to find out the state of a thread. It returns true if a thread is in the Ready, Blocked, or Running state; it returns false if a thread is new and has not started or if it is finished.

The interrupt() method interrupts a thread in the following way: If a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.

The isInterrupt() method tests whether the thread is interrupted.

The deprecated stop(), suspend(), and resume() Methods

NOTE: The <u>Thread</u> class also contains the <u>stop()</u>, <u>suspend()</u>, and <u>resume()</u> methods. As of Java 2, these methods are *deprecated* (or *outdated*) because they are known to be inherently unsafe. You should assign <u>null</u> to a <u>Thread</u> variable to indicate that it is stopped rather than use the <u>stop()</u> method.

Thread Priority

☞ Each thread is assigned a default priority of `Thread.NORM_PRIORITY`. You can reset the priority using `setPriority(int priority).`

☞ Some constants for priorities include
`Thread.MIN_PRIORITY`
`Thread.MAX_PRIORITY`
`Thread.NORM_PRIORITY`

Example: Flashing Text
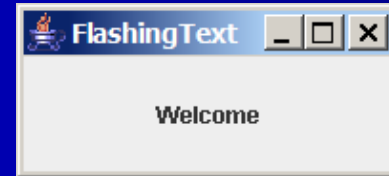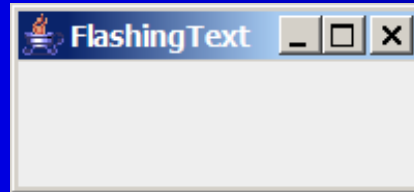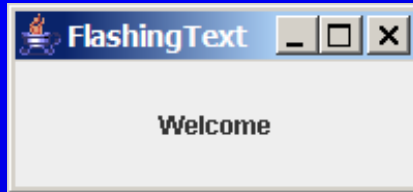
FlashingText

GUI Event Dispatcher Thread

GUI event handling and painting code executes in a single thread, called the *event dispatcher thread*. This ensures that each event handler finishes executing before the next one executes and the painting isn't interrupted by events.

Launch Application from Main Method

So far, you have launched your GUI application from the main method by creating a frame and making it visible. This works fine for most applications. In certain situations, however, it could cause problems. To avoid possible thread deadlock, you should launch GUI creation from the event dispatcher thread as follows:

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            // Place the code for creating a frame and setting it properties
        }
    });
}
```

invokeLater and invokeAndWait

In certain situations, you need to run the code in the event dispatcher thread to avoid possible deadlock. You can use the static methods, invokeLater and invokeAndWait, in the javax.swing.SwingUtilities class to run the code in the event dispatcher thread. You must put this code in the run method of a Runnable object and specify the Runnable object as the argument to invokeLater and invokeAndWait. The invokeLater method returns immediately, without waiting for the event dispatcher thread to execute the code. The invokeAndWait method is just like invokeLater, except that invokeAndWait doesn't return until the event-dispatching thread has executed the specified code.

GUI Event Dispatcher Thread Demo

EventDispatcherThreadDemo

Case Study: Clock with Audio (Optional)
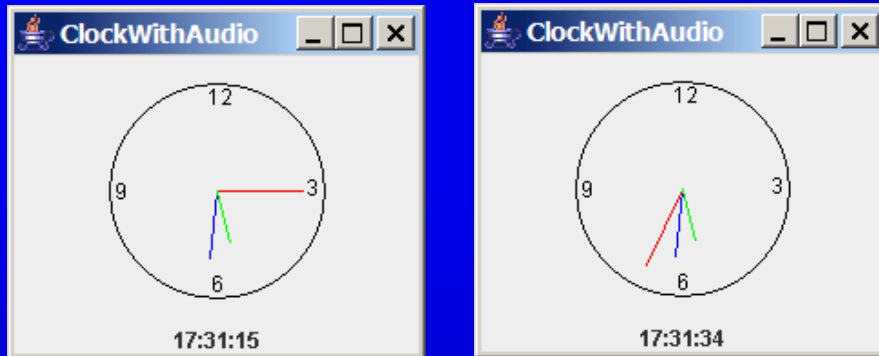
The example creates an applet that displays a running clock and announces the time at one-minute intervals. For example, if the current time is 6:30:00, the applet announces, "six o'clock thirty minutes a.m." If the current time is 20:20:00, the applet announces, "eight o'clock twenty minutes p.m." Also add a label to display the digital time.

ClockWithAudio

Run Audio on Separate Thread

When you run the preceding program, you will notice that the second hand does not display at the first, second, and third seconds of the minute. This is because sleep(1500) is invoked twice in the announceTime() method, which takes three seconds to announce the time at the beginning of each minute. Thus, the next action event is delayed for three seconds during the first three seconds of each minute. As a result of this delay, the time is not updated and the clock was not repainted for these three seconds. To fix this problem, you should announce the time on a separate thread. This can be accomplished by modifying the announceTime method.

ClockWithAudioOnSeparateThread

Thread Pools

Starting a new thread for each task could limit throughput and cause poor performance. A thread pool is ideal to manage the number of tasks executing concurrently. JDK 1.5 uses the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks. ExecutorService is a subinterface of Executor.

«interface» *java.util.concurrent.Executor*	
+execute(Runnable object): void	Executes the runnable task.

«interface» *java.util.concurrent.ExecutorService*	
+shutdown(): void	Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.
+shutdownNow(): List<Runnable>	Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.
+isShutdown(): boolean	Returns true if the executor has been shutdown.
+isTerminated(): boolean	Returns true if all tasks in the pool are terminated.

Creating Executors

To create an <u>Executor</u> object, use the static methods in the <u>Executors</u> class.

java.util.concurrent.Executors	
+newFixedThreadPool(numberOfThreads: int): ExecutorService	Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.
+newCachedThreadPool(): ExecutorService	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

<u>ExecutorDemo</u>

Thread Synchronization

A shared resource may be corrupted if it is accessed simultaneously by multiple threads. For example, two unsynchronized threads accessing the same bank account may cause conflict.

Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

Example: Showing Resource Conflict

☞ Objective: Write a program that demonstrates the problem of resource conflict. Suppose that you create and launch one hundred threads, each of which adds a penny to an account. Assume that the account is initially empty.
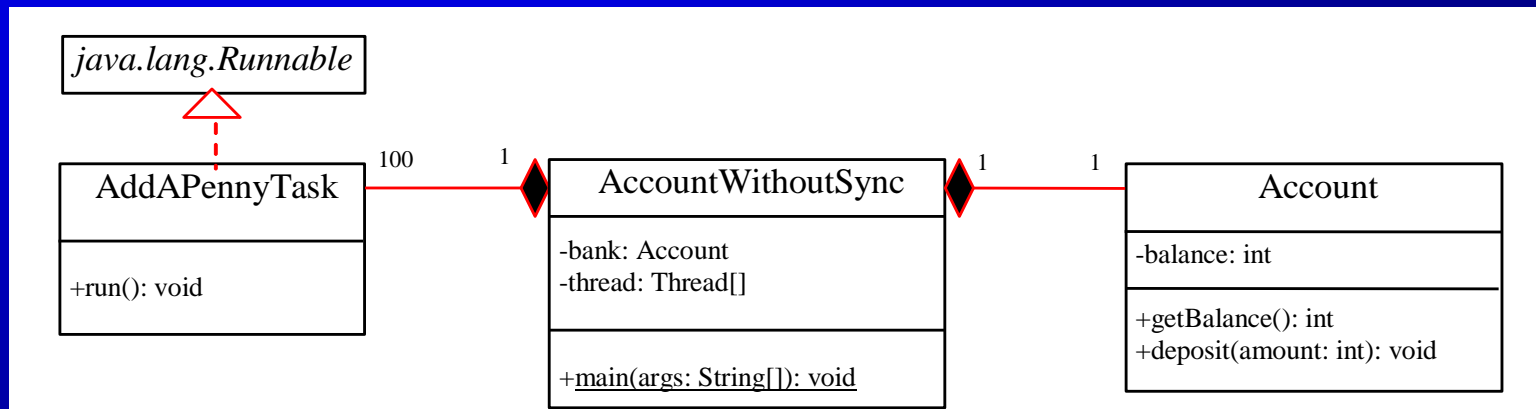
AccountWithoutSync

Race Condition

What, then, caused the error in the example? Here is a possible scenario:

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

The effect of this scenario is that Task 1 did nothing, because in Step 4 Task 2 overrides Task 1's result. Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes conflict. This is a common problem known as a *race condition* in multithreaded programs. A class is said to be *thread-safe* if an object of the class does not cause a race condition in the presence of multiple threads. As demonstrated in the preceding example, the Account class is not thread-safe.

The `synchronized` keyword

To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as critical region. The critical region in the Listing 29.7 is the entire deposit method. You can use the synchronized keyword to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the problem in Listing 29.7, one approach is to make Account thread-safe by adding the synchronized keyword in the deposit method in Line 45 as follows:

```
public synchronized void deposit(double amount)
```

Synchronizing Instance Methods and Static Methods

A synchronized method acquires a lock before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class. If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

Synchronizing Instance Methods and Static Methods

With the deposit method synchronized, the preceding scenario cannot happen. If Task 2 starts to enter the method, and Task 1 is already in the method, Task 2 is blocked until Task 1 finishes the method.

Task 1	Task 2
Acquire a lock on the object account	
↓	
Execute the deposit method	
↓	Wait to acquire the lock
Release the lock	↓
	Acqurie a lock on the object account
	↓
	Execute the deposit method
	↓
	Release the lock
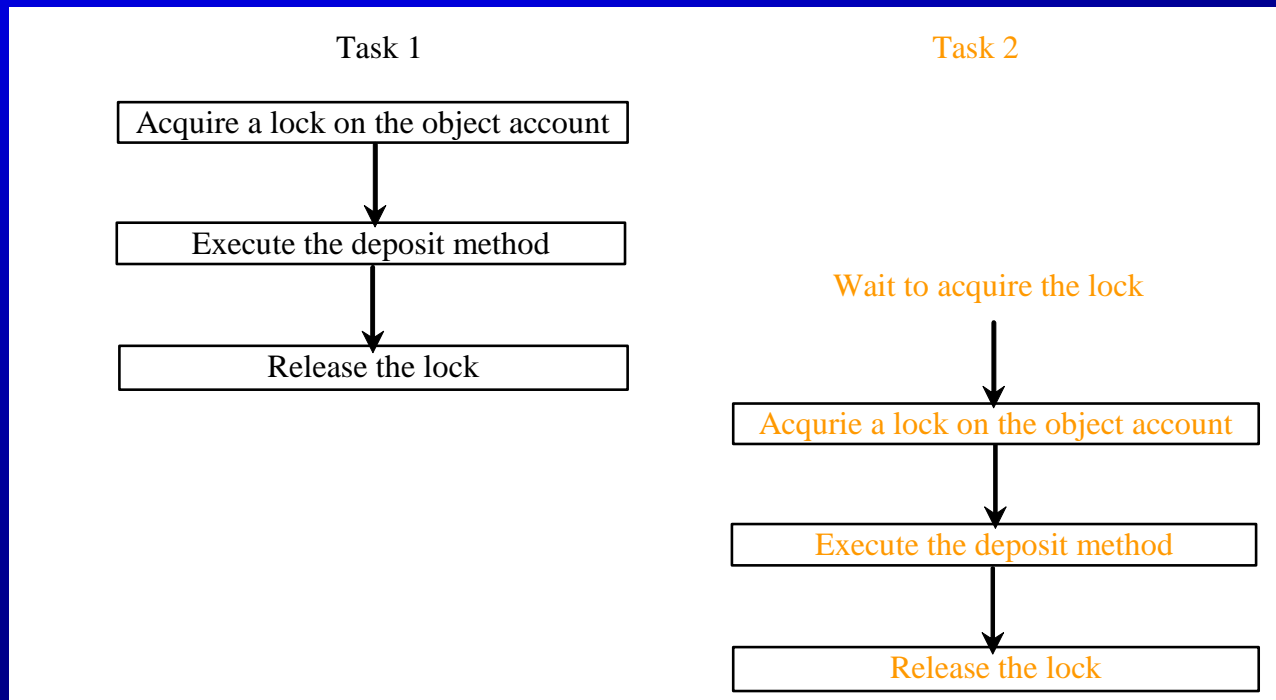
Synchronizing Statements

Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class. A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method. This block is referred to as a *synchronized block*. The general form of a synchronized statement is as follows:

```
synchronized (expr) {
   statements;
}
```

The expression expr must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released. When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronizing Statements vs. Methods

Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {
  // method body
}
```
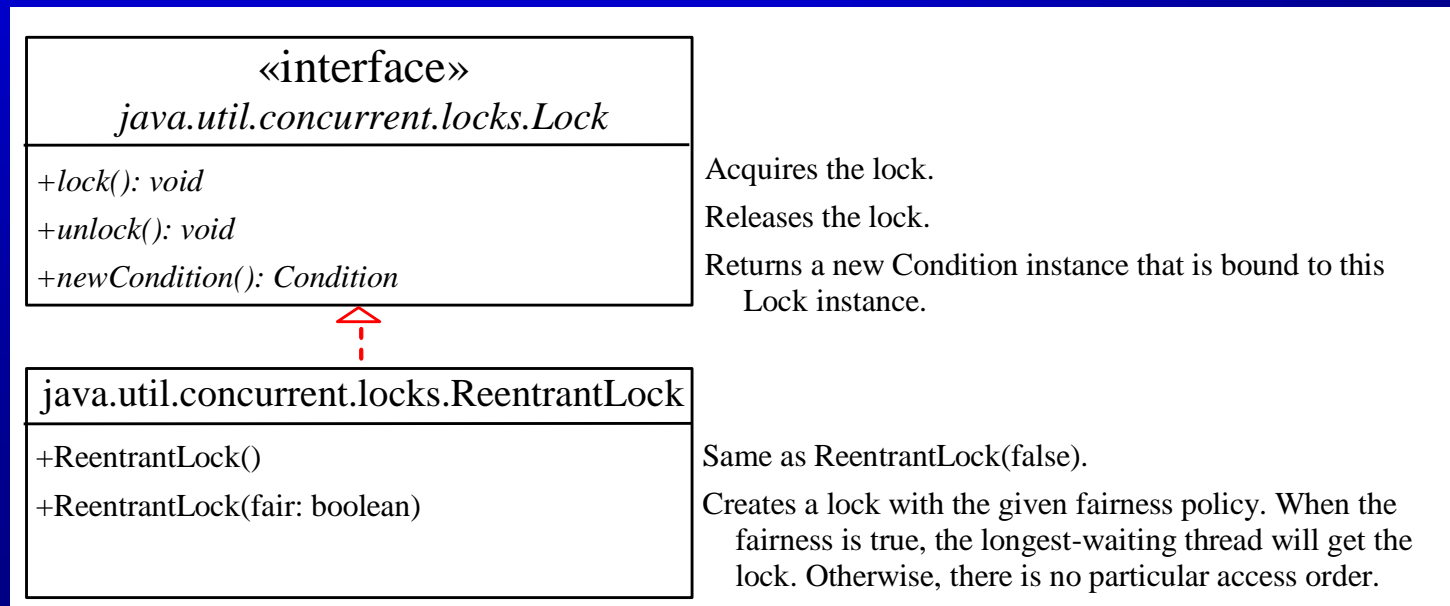
This method is equivalent to

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

Synchronization Using Locks

A synchronized instance method implicitly acquires a lock on the instance before it executes the method.
JDK 1.5 enables you to use locks explicitly. The new locking features are flexible and give you more control for coordinating threads. A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks, as shown in Figure 29.14. A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.

«interface» *java.util.concurrent.locks.Lock*	
+*lock(): void*	Acquires the lock.
+*unlock(): void*	Releases the lock.
+*newCondition(): Condition*	Returns a new Condition instance that is bound to this Lock instance.

java.util.concurrent.locks.ReentrantLock	
+ReentrantLock()	Same as ReentrantLock(false).
+ReentrantLock(fair: boolean)	Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

Fairness Policy

ReentrantLock is a concrete implementation of Lock for creating mutual exclusive locks. You can create a lock with the specified fairness policy. True fairness policies guarantee the longest-wait thread to obtain the lock first. False fairness policies grant a lock to a waiting thread without any access order. Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Example: Using Locks

This example revises AccountWithoutSync.java in Listing 29.7 to synchronize the account modification using explicit locks.

AccountWithSyncUsingLock

Cooperation Among Threads

The conditions can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. Conditions are objects created by invoking the <u>newCondition()</u> method on a <u>Lock</u> object. Once a condition is created, you can use its <u>await()</u>, <u>signal()</u>, and <u>signalAll()</u> methods for thread communications, as shown in Figure 29.15. The <u>await()</u> method causes the current thread to wait until the condition is signaled. The <u>signal()</u> method wakes up one waiting thread, and the <u>signalAll()</u> method wakes all waiting threads.

«interface» java.util.concurrent.Condition	
+*await(): void*	Causes the current thread to wait until the condition is signaled.
+*signal(): void*	Wakes up one waiting thread.
+*signalAll(): Condition*	Wakes up all waiting threads.
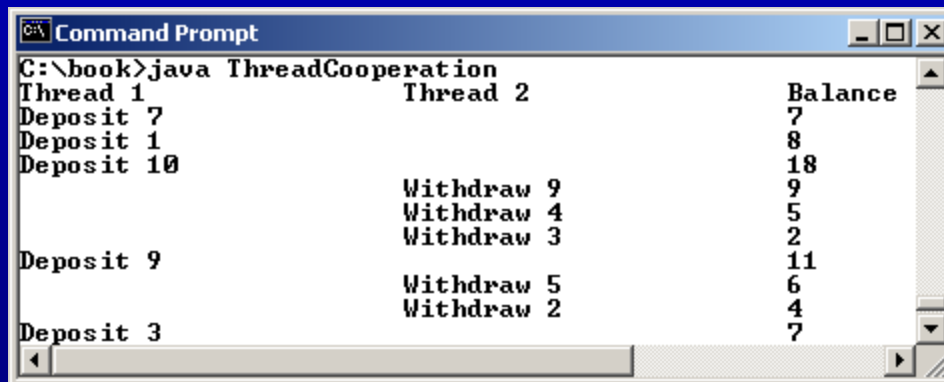
Cooperation Among Threads

To synchronize the operations, use a lock with a condition: newDeposit (i.e., new deposit added to the account). If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again. The interaction between the two tasks is shown in Figure 29.16.

Withdraw Task

```
lock.lock();
```

```
while (balance < withdrawAmount)
    newDeposit.await();
```

```
balance -= withdrawAmount
```

```
lock.unlock();
```

Deposit Task

```
lock.lock();
```

```
balance += depositAmount
```

```
newDeposit.signalAll();
```

```
lock.unlock();
```
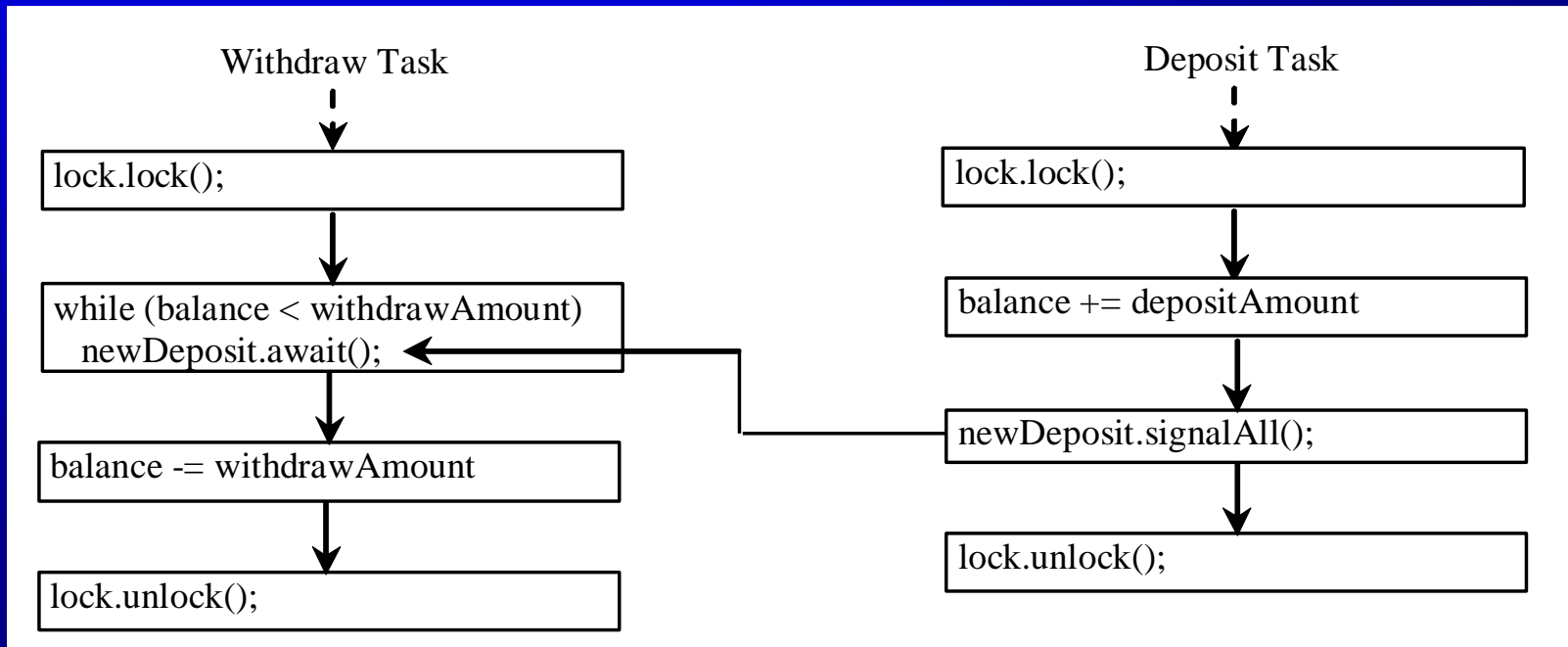
Example: Thread Cooperation

Write a program that demonstrates thread cooperation. Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund is deposited to the account, the first thread notifies the second thread to resume. If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account. Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.

```
Command Prompt                                    _ □ ×
C:\book>java ThreadCooperation
Thread 1                Thread 2             Balance
Deposit 7                                    7
Deposit 1                                    8
Deposit 10                                   18
                        Withdraw 9           9
                        Withdraw 4           5
                        Withdraw 3           2
Deposit 9                                    11
                        Withdraw 5           6
                        Withdraw 2           4
Deposit 3                                    7
```

ThreadCooperation

Java's Built-in Monitors (Optional)

Locks and conditions are new in Java 5. Prior to Java 5, thread communications are programmed using object's built-in monitors. Locks and conditions are more powerful and flexible than the built-in monitor. For this reason, this section can be completely ignored. However, if you work with legacy Java code, you may encounter the Java's built-in monitor. A *monitor* is an object with mutual exclusion and synchronization capabilities. Only one thread can execute a method at a time in the monitor. A thread enters the monitor by acquiring a lock on the monitor and exits by releasing the lock. *Any object can be a monitor*. An object becomes a monitor once a thread locks it. Locking is implemented using the <u>synchronized</u> keyword on a method or a block. A thread must acquire a lock before executing a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.

wait(), notify(), and notifyAll()

Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.

The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

Example: Using Monitor

Task 1   Task 2

```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();          ←── resume

    // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
```
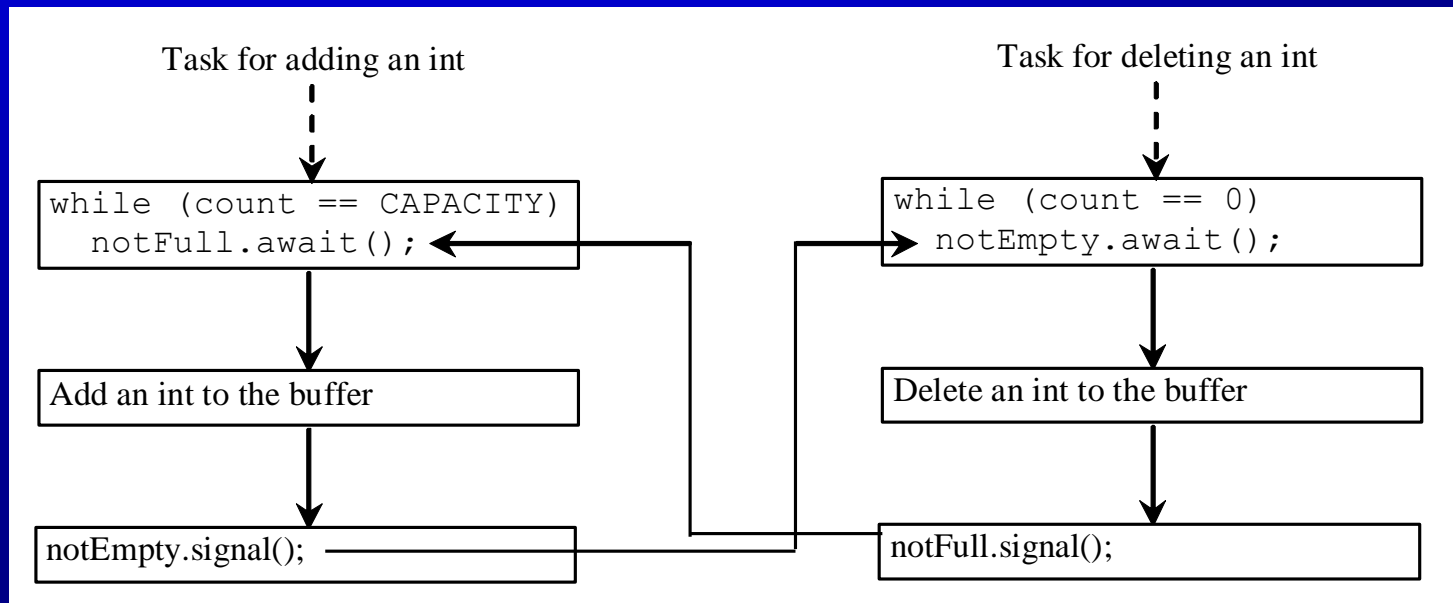
```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify();  or  anObject.notifyAll();
  ...
}
```

☞ The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an IllegalMonitorStateException will occur.

☞ When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.

☞ The wait(), notify(), and notifyAll() methods on an object are analogous to the await(), signal(), and signalAll() methods on a condition.

Case Study: Producer/Consumer (Optional)

Consider the classic Consumer/Producer example. Suppose you use a buffer to store integers. The buffer size is limited. The buffer provides the method write(int) to add an int value to the buffer and the method read() to read and delete an int value from the buffer. To synchronize the operations, use a lock with two conditions: notEmpty (i.e., buffer is not empty) and notFull (i.e., buffer is not full). When a task adds an int to the buffer, if the buffer is full, the task will wait for the notFull condition. When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the notEmpty condition. The interaction between the two tasks is shown in Figure 29.19.

Task for adding an int

Task for deleting an int

```
while (count == CAPACITY)
  notFull.await();
```

```
while (count == 0)
  notEmpty.await();
```

Add an int to the buffer

Delete an int to the buffer

notEmpty.signal();

notFull.signal();
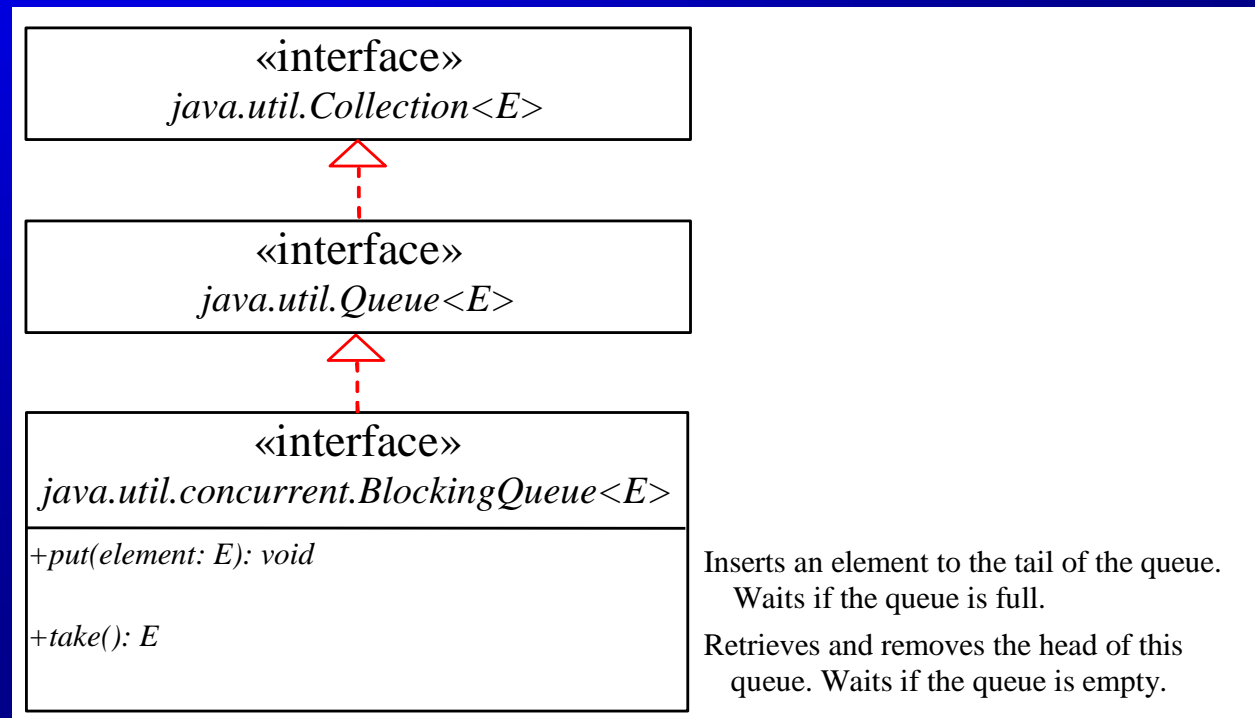
Case Study: Producer/Consumer (Optional)

Listing 29.10 presents the complete program. The program contains the <u>Buffer</u> class (lines 43-89) and two tasks for repeatedly producing and consuming numbers to and from the buffer (lines 15-41). The <u>write(int)</u> method (line 58) adds an integer to the buffer. The <u>read()</u> method (line 75) deletes and returns an integer from the buffer.

For simplicity, the buffer is implemented using a linked list (lines 48-49). Two conditions <u>notEmpty</u> and <u>notFull</u> on the lock are created in lines 55-56. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the <u>wait()</u> and <u>notify()</u> methods to rewrite this example, you have to designate two objects as monitors.

<u>ConsumerProducer</u>

Blocking Queues (Optional)

§ 22.8 introduced queues and priority queues. A *blocking queue* causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue.

«interface»
java.util.Collection<E>

«interface»
java.util.Queue<E>

«interface»
java.util.concurrent.BlockingQueue<E>

+*put(element: E): void*

+*take(): E*

Inserts an element to the tail of the queue. Waits if the queue is full.

Retrieves and removes the head of this queue. Waits if the queue is empty.

Concrete Blocking Queues

Three concrete blocking queues ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are supported in JDK 1.5, as shown in Figure 29.22. All are in the java.util.concurrent package. ArrayBlockingQueue implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an ArrayBlockingQueue. LinkedBlockingQueue implements a blocking queue using a linked list. You may create an unbounded or bounded LinkedBlockingQueue. PriorityBlockingQueue is a priority queue. You may create an unbounded or bounded priority queue.

«interface»
java.util.concurrent.BlockingQueue\<E\>

ArrayBlockingQueue\<E\>
+ArrayBlockingQueue(capacity: int)
+ArrayBlockingQueue(capacity: int, fair: boolean)

LinkedBlockingQueue\<E\>
+LinkedBlockingQueue()
+LinkedBlockingQueue(capacity: int)

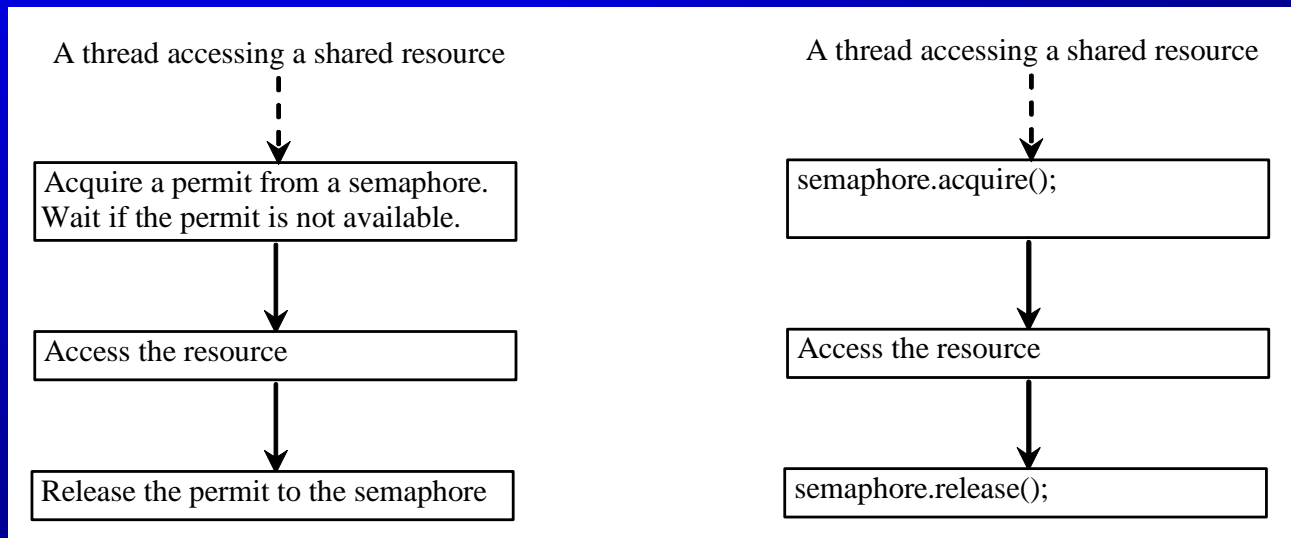| PriorityBlockingQueue\<E\> |
|---|
| +PriorityBlockingQueue() |
| +PriorityBlockingQueue(capacity: int) |
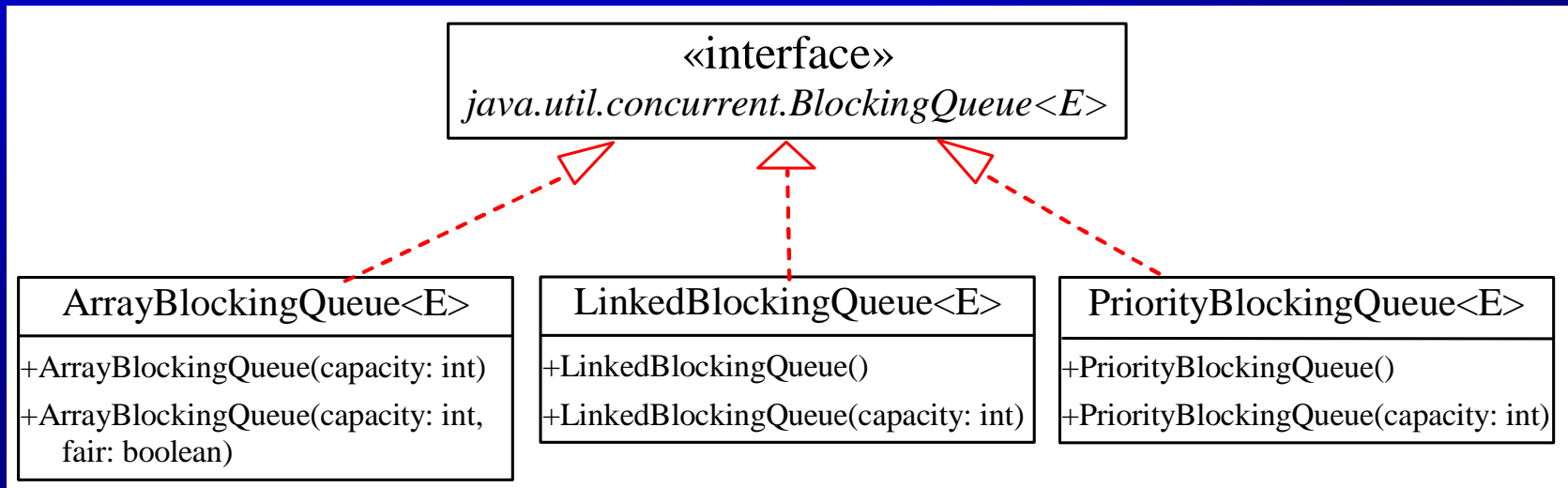
Producer/Consumer Using Blocking Queues

Listing 29.11 gives an example of using an <u>ArrayBlockingQueue</u> to simplify the Consumer/Producer example in Listing 29.11.

<u>ConsumerProducerUsingBlockingQueue</u>

Semaphores (Optional)

Semaphores can be used to restrict the number of threads that access a shared resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore, as shown in Figure 29.29.

A thread accessing a shared resource

Acquire a permit from a semaphore. Wait if the permit is not available.

Access the resource

Release the permit to the semaphore

A thread accessing a shared resource

semaphore.acquire();

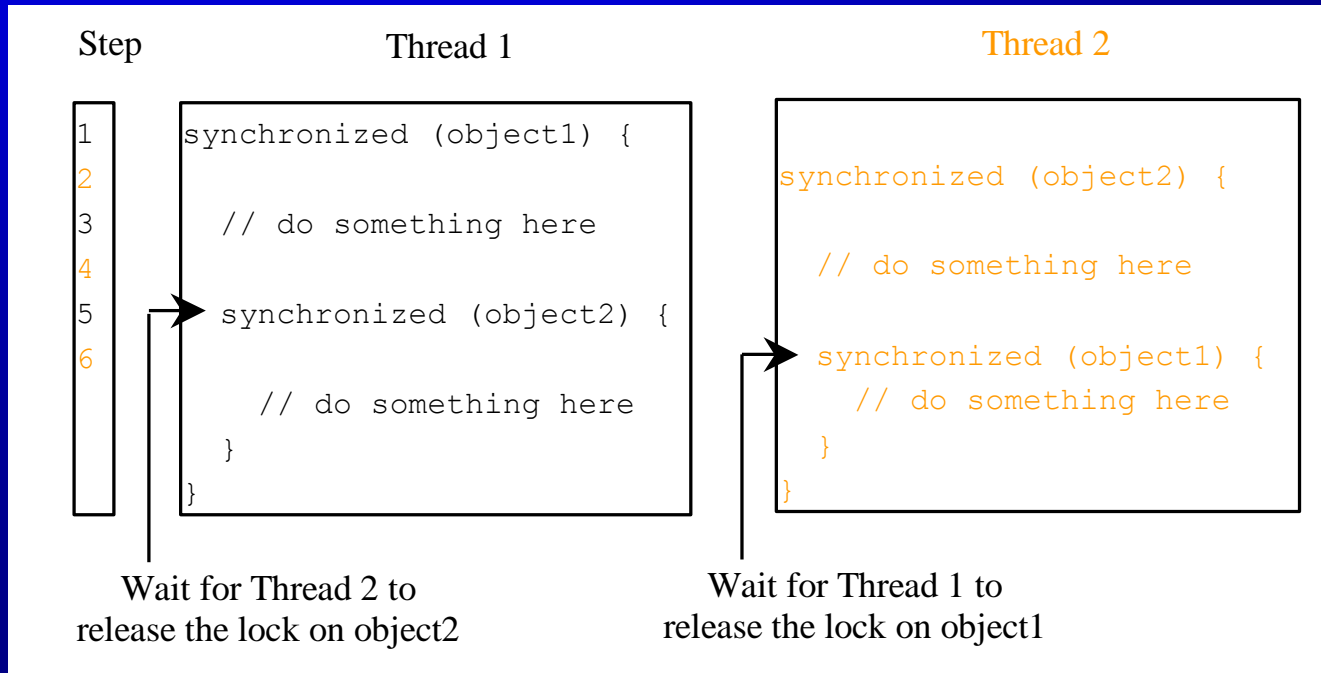Access the resource

semaphore.release();

Creating Semaphores

To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in Figure 29.29. A task acquires a permit by invoking the semaphore's <u>acquire()</u> method and releases the permit by invoking the semaphore's <u>release()</u> method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1. Once a permit is released, the total number of available permits in a semaphore is increased by 1.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

Deadlock

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects, as shown in Figure 29.15. Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2. Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1. The two threads wait for each other to release the in order to get the lock, and neither can continue to run.

Step	Thread 1	Thread 2
1	`synchronized (object1) {`	
2		`synchronized (object2) {`
3	`  // do something here`	
4		`  // do something here`
5	`  synchronized (object2) {`	
6		`  synchronized (object1) {`
	`    // do something here`	`    // do something here`
	`  }`	`  }`
	`}`	`}`

Wait for Thread 2 to release the lock on object2

Wait for Thread 1 to release the lock on object1

Preventing Deadlock

Deadlock can be easily avoided by using a simple technique known as resource ordering. With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order. For the example in Figure 29.15, suppose the objects are ordered as object1 and object2. Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2. Once Thread 1 acquired a lock on object1, Thread 2 has to wait for a lock on object1. So Thread 1 will be able to acquire a lock on object2 and no deadlock would occur.

Thread States

A thread can be in one of five states:
New, Ready, Running, Blocked, or
Finished.
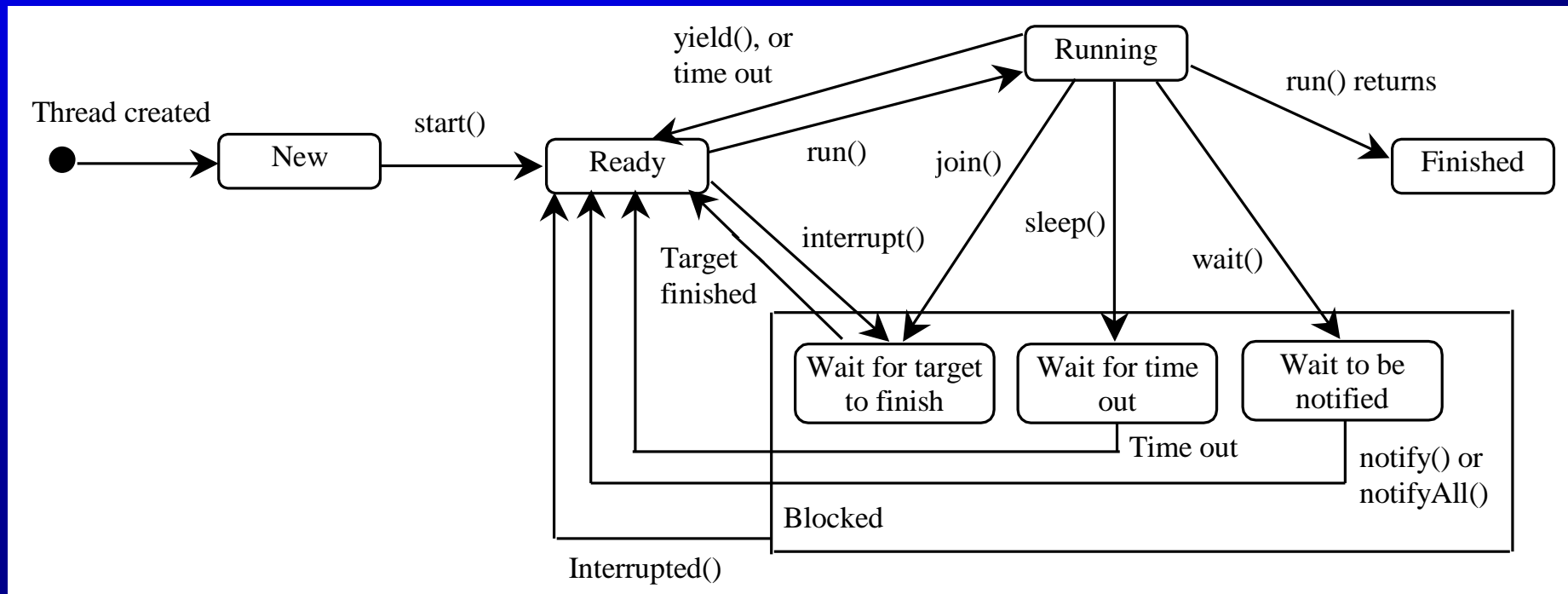
Synchronized Collections

The classes in the Java Collections Framework are not thread-safe, i.e., the contents may be corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or using synchronized collections.

The Collections class provides six static methods for wrapping a collection into a synchronized version, as shown in Figure 29.27. The collections created using these methods are called *synchronization wrappers*.

java.util.Collections	
+synchronizedCollection(c: Collection): Collection	Returns a synchronized collection.
+synchronizedList(list: List): List	Returns a synchronized list from the specified list.
+synchronizedMap(m: Map): Map	Returns a synchronized map from the specified map.
+synchronizedSet(s: Set): Set	Returns a synchronized set from the specified set.
+synchronizedSortedMap(s: SortedMap): SortedMap	Returns a synchronized sorted map from the specified sorted map.
+synchronizedSortedSet(s: SortedSet): SortedSet	Returns a synchronized sorted set.

Vector, Stack, and Hashtable

Invoking synchronizedCollection(Collection c) returns a new Collection object, in which all the methods that access and update the original collection c are synchronized. These methods are implemented using the synchronized keyword. For example, the add method is implemented like this:

```
public boolean add(E o) {
  synchronized (this) { return c.add(o); }
}
```

The synchronized collections can be safely accessed and modified by multiple threads concurrently.

The methods in java.util.Vector, java.util.Stack, and Hashtable are already synchronized. These are old classes introduced in JDK 1.0. In JDK 1.5, you should use java.util.ArrayList to replace Vector, java.util.LinkedList to replace Stack, and java.util.Map to replace Hashtable. If synchronization is needed, use a synchronization wrapper.