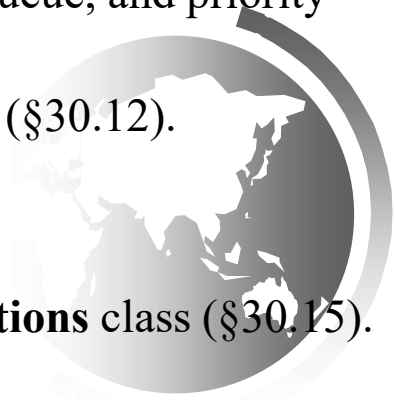
Chapter 30 Multithreading and Parallel Programming

Objectives

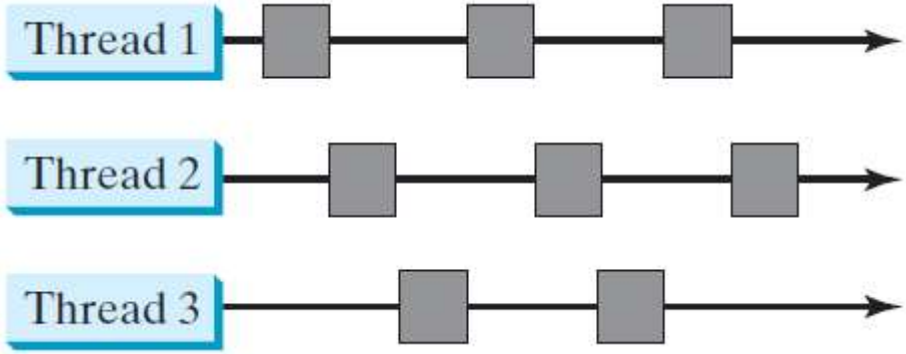
- To get an overview of multithreading (§30.2).
- To develop task classes by implementing the **Runnable** interface (§30.3).
- To create threads to run tasks using the **Thread** class (§30.3).
- To control threads using the methods in the **Thread** class (§30.4).
- To control animations using threads and use **Platform.runLater** to run the code in application thread (§30.5).
- To execute tasks in a thread pool (§30.6).
- To use synchronized methods or blocks to synchronize threads to avoid race conditions (§30.7).
- To synchronize threads using locks (§30.8).
- To facilitate thread communications using conditions on locks (§§30.9–30.10).
- To use blocking queues to synchronize access to an array queue, linked queue, and priority queue (§30.11).
- To restrict the number of accesses to a shared resource using semaphores (§30.12).
- To use the resource-ordering technique to avoid deadlocks (§30.13).
- To describe the life cycle of a thread (§30.14).
- To create synchronized collections using the static methods in the **Collections** class (§30.15).
- To develop parallel programs using the Fork/Join Framework (§30.16).



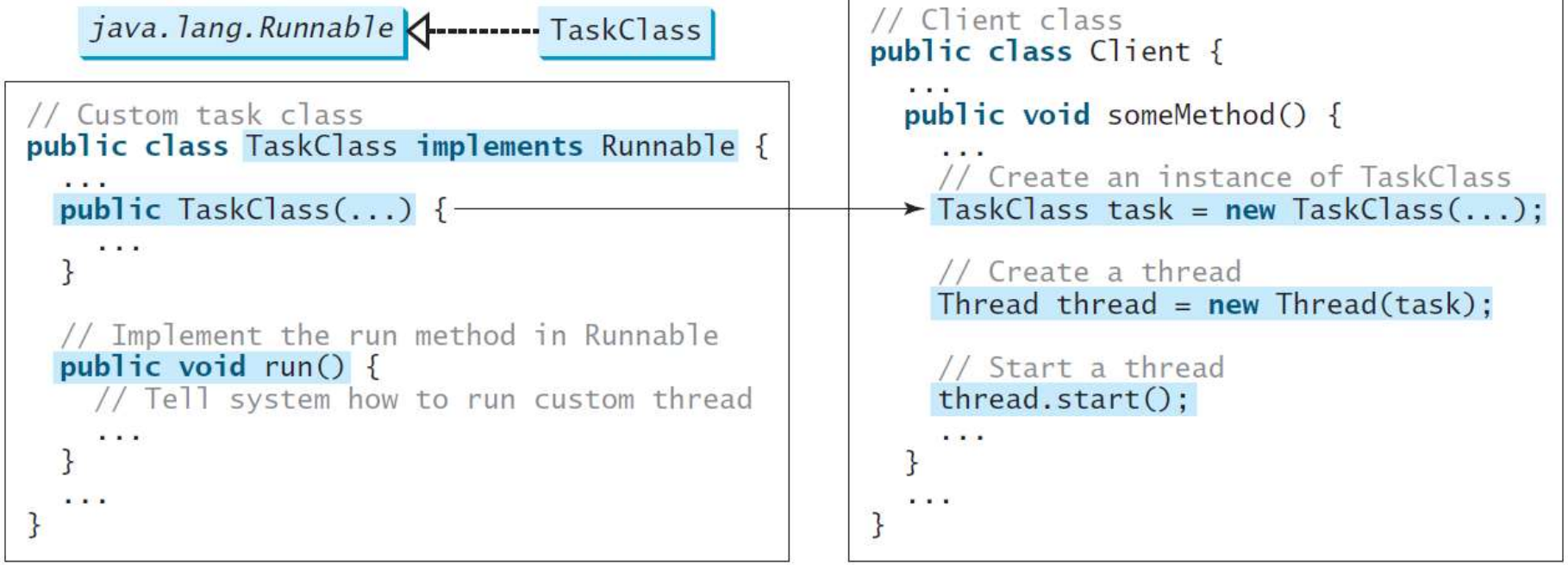
Threads Concept

Multiple threads on multiple CPUs

Multiple threads sharing a single CPU



Creating Tasks and Threads

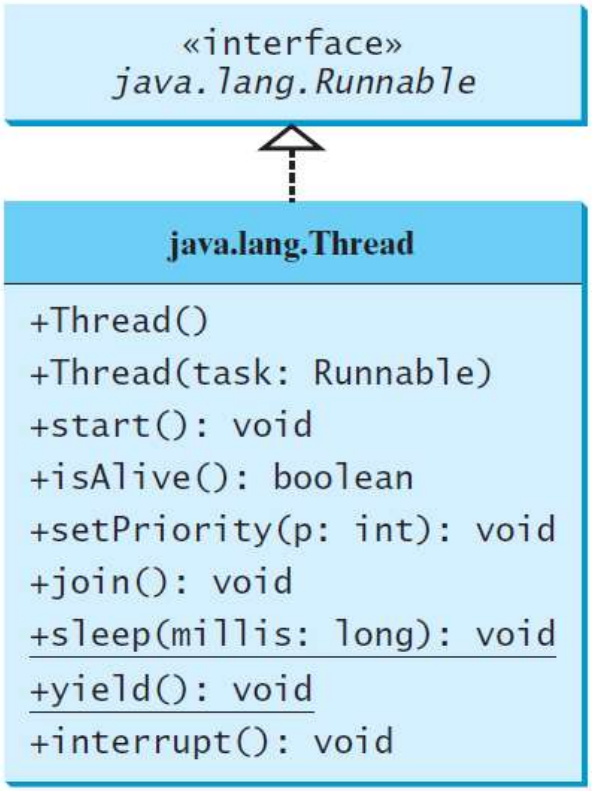


Example: Using the `Runnable` Interface to Create and Launch Threads

- 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 Thread Class



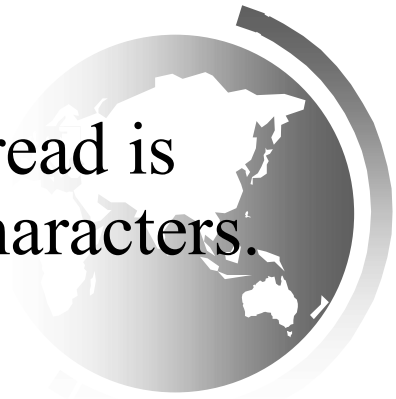
`+Thread()`	Creates an empty thread.
`+Thread(task: Runnable)`	Creates a thread for a specified task.
`+start(): void`	Starts the thread that causes the `run()` method to be invoked by the JVM.
`+isAlive(): boolean`	Tests whether the thread is currently running.
`+setPriority(p: int): void`	Sets priority `p` (ranging from 1 to 10) for this thread.
`+join(): void`	Waits for this thread to finish.
`+sleep(millis: long): void`	Puts a thread to sleep for a specified time in milliseconds.
`+yield(): void`	Causes a thread to pause temporarily 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:

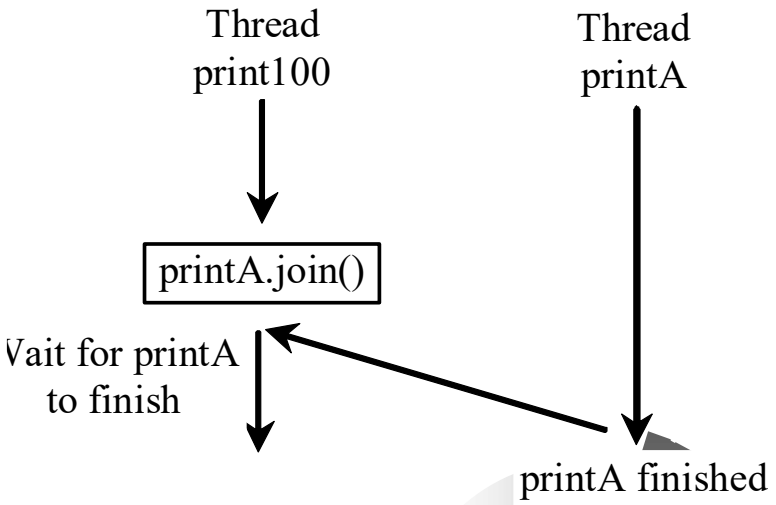
```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

Every time a number (>= 50) is printed, the print100 thread is put to sleep for 1 millisecond.

The 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:

```
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) {
  }
```



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

Thread Priority

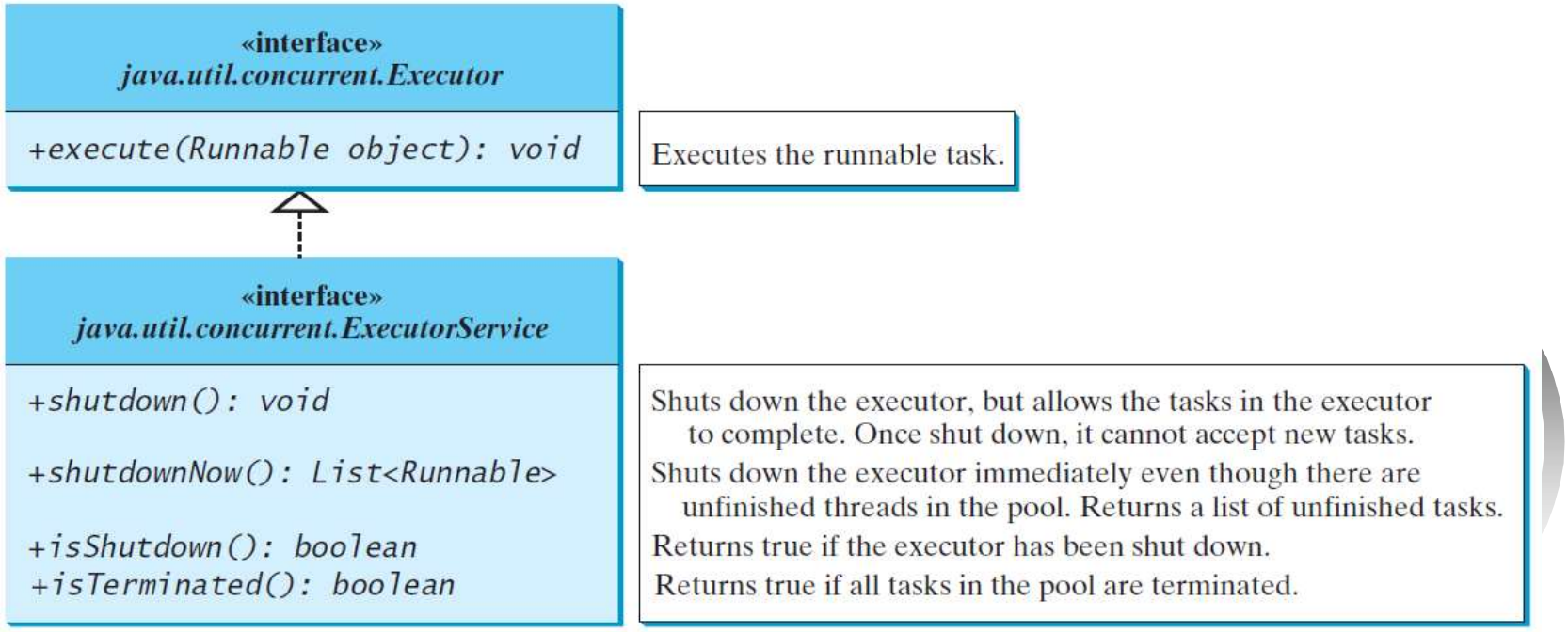
- Each thread is assigned a default priority of `Thread.NORM_PRIORITY`. You can reset the priority using `setPriority(int priority)`.

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

Example: Flashing Text

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.



Creating Executors

To create an Executor object, use the static methods in the Executors 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.

ExecutorDemo Run

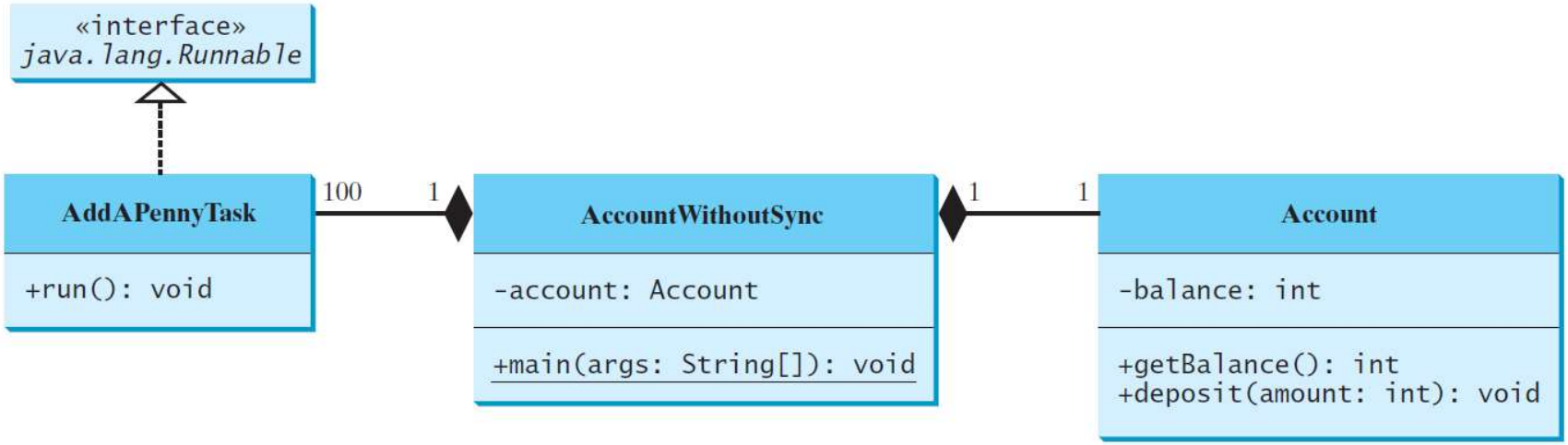
Thread Synchronization

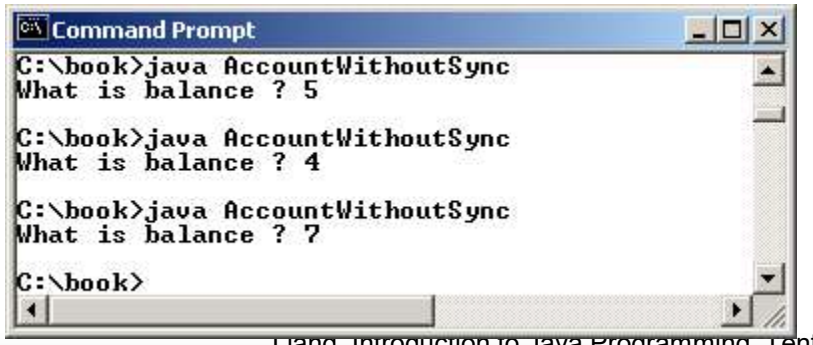
A shared resource may be corrupted if it is accessed simultaneously by multiple threads. For example, two unsynchronized threads accessing the same bank account may cause conflict.

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 Run

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 30.5 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 30.5, 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

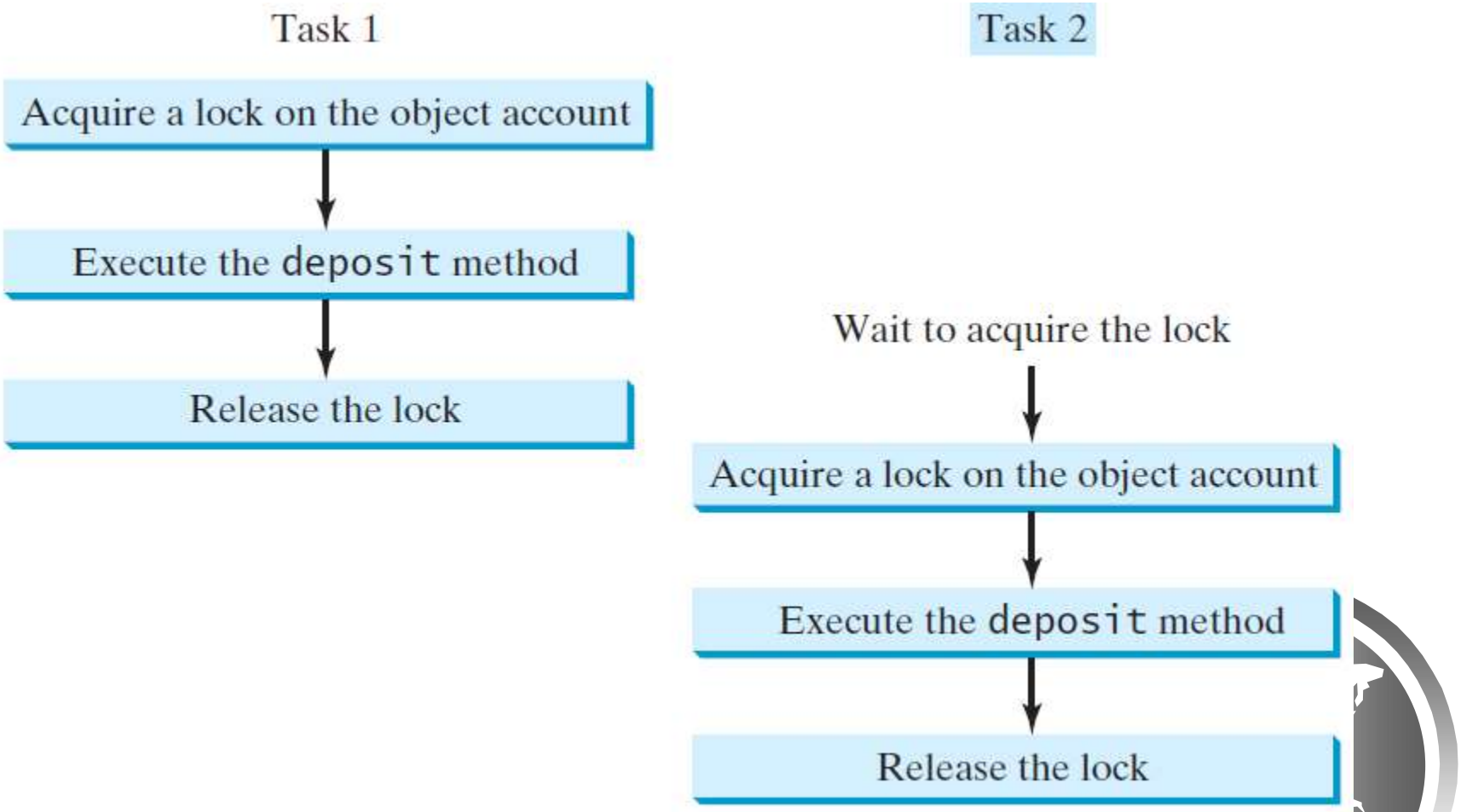
A synchronized method acquires a lock before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class. If one thread invokes a synchronized instance method (respectively, static method) on an object, 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.

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

Synchronizing Tasks

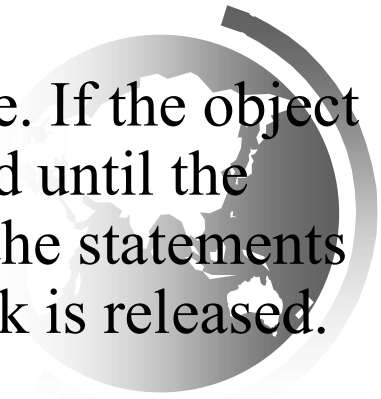


Synchronizing Statements

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
}
```

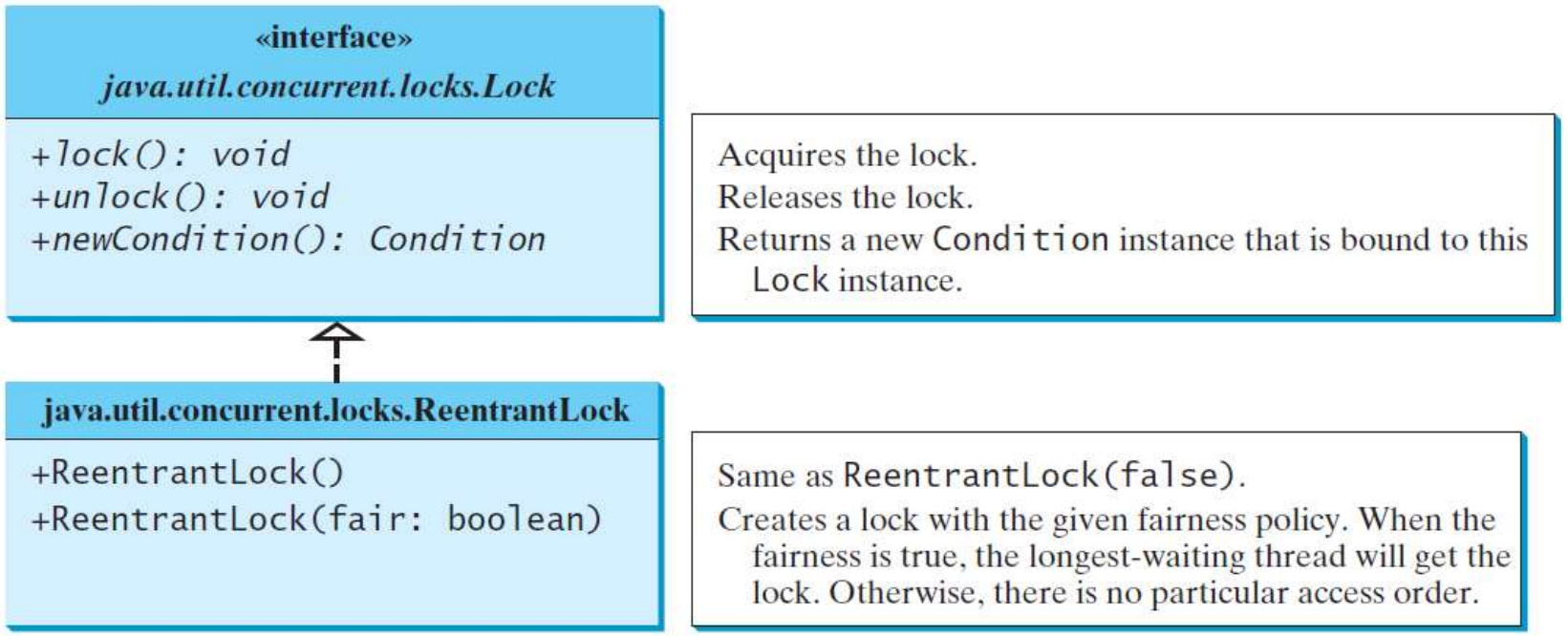
This method is equivalent to

```
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. A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.

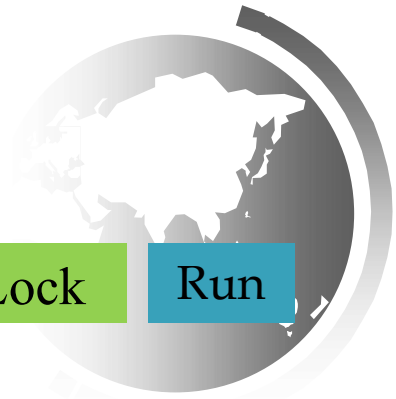


Fairness Policy

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

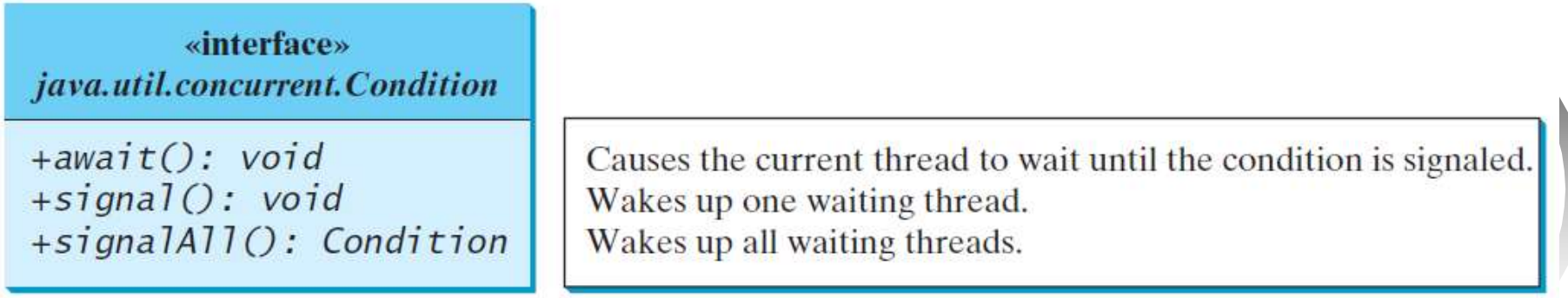
This example revises AccountWithoutSync.java to synchronize the account modification using explicit locks.



AccountWithSyncUsingLock Run

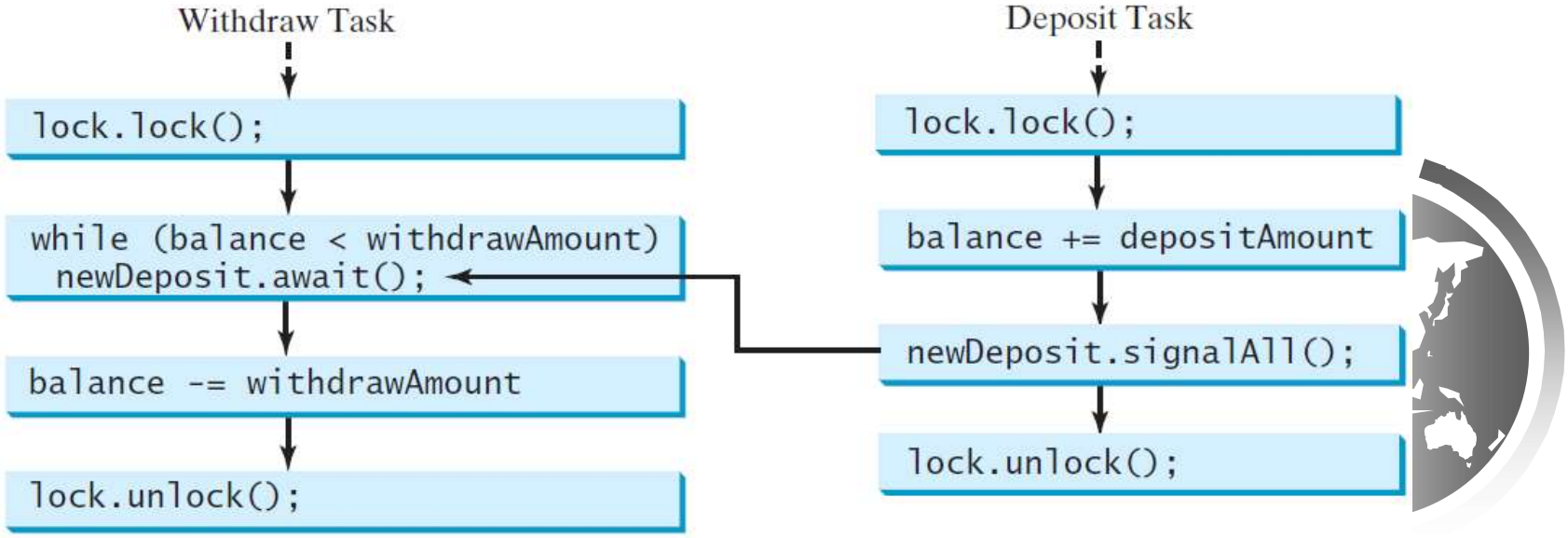
Cooperation Among Threads

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 newCondition() method on a Lock object. Once a condition is created, you can use its await(), signal(), and signalAll() methods for thread communications. The await() method causes the current thread to wait until the condition is signaled. The signal() method wakes up one waiting thread, and the signalAll() method wakes 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.



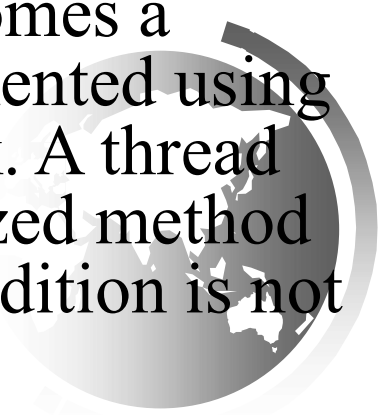
Example: Thread Cooperation

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.

ThreadCooperation Run

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 synchronized 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

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

    // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

resume

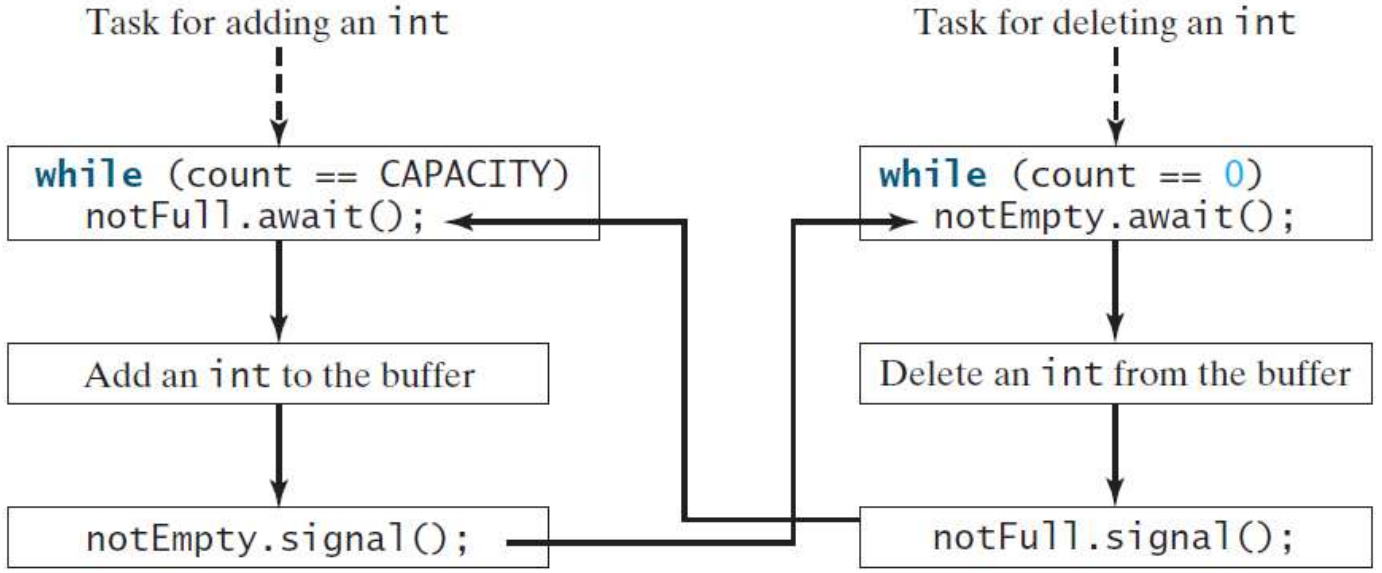
Task 2

```
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.



Case Study: Producer/Consumer (Optional)

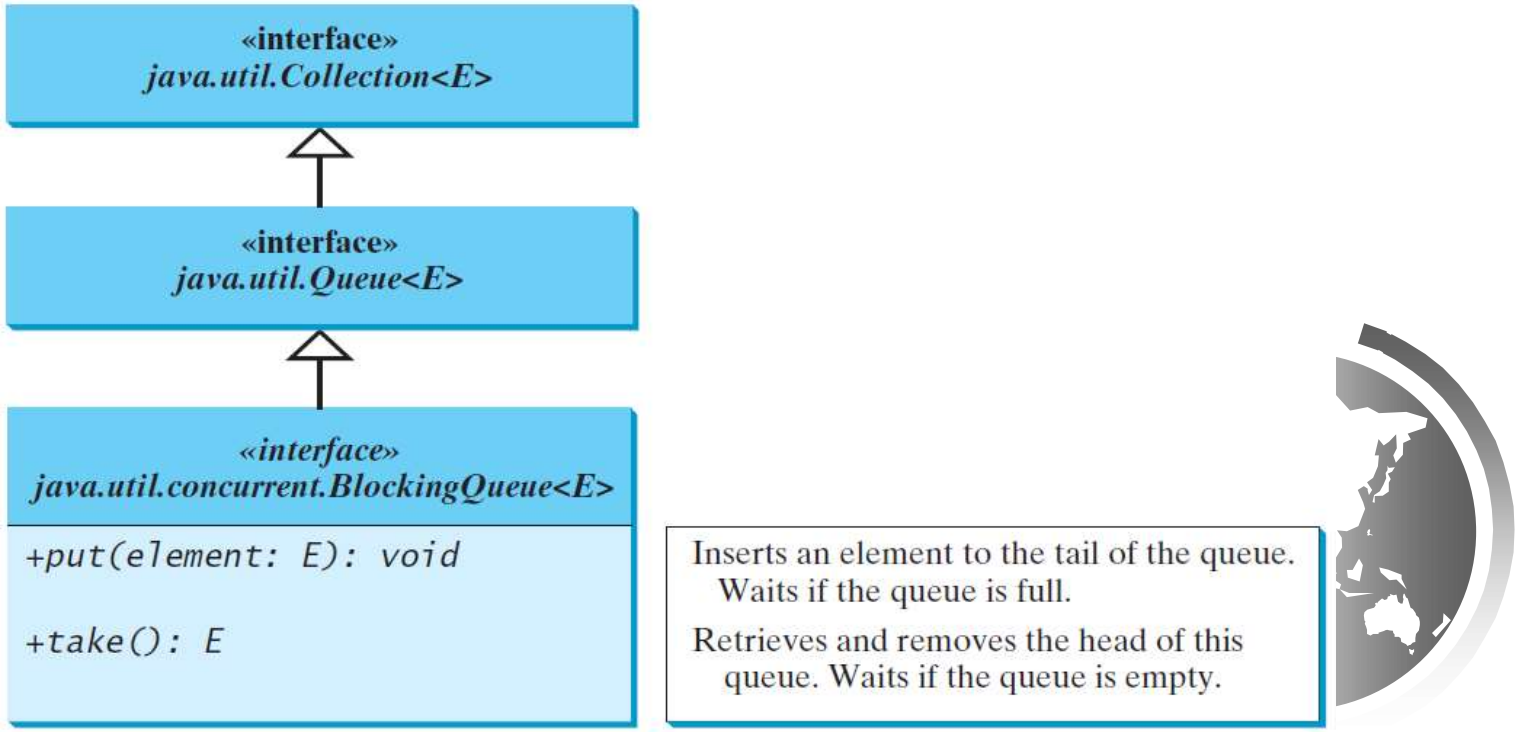
Listing 30.8 presents the complete program. The program contains the Buffer class (lines 43-89) and two tasks for repeatedly producing and consuming numbers to and from the buffer (lines 15-41). The write(int) method (line 58) adds an integer to the buffer. The read() method (line 75) deletes and returns an integer from the buffer.

For simplicity, the buffer is implemented using a linked list (lines 48-49). Two conditions notEmpty and notFull 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 wait() and notify() methods to rewrite this example, you have to designate two objects as monitors.

ConsumerProducer Run

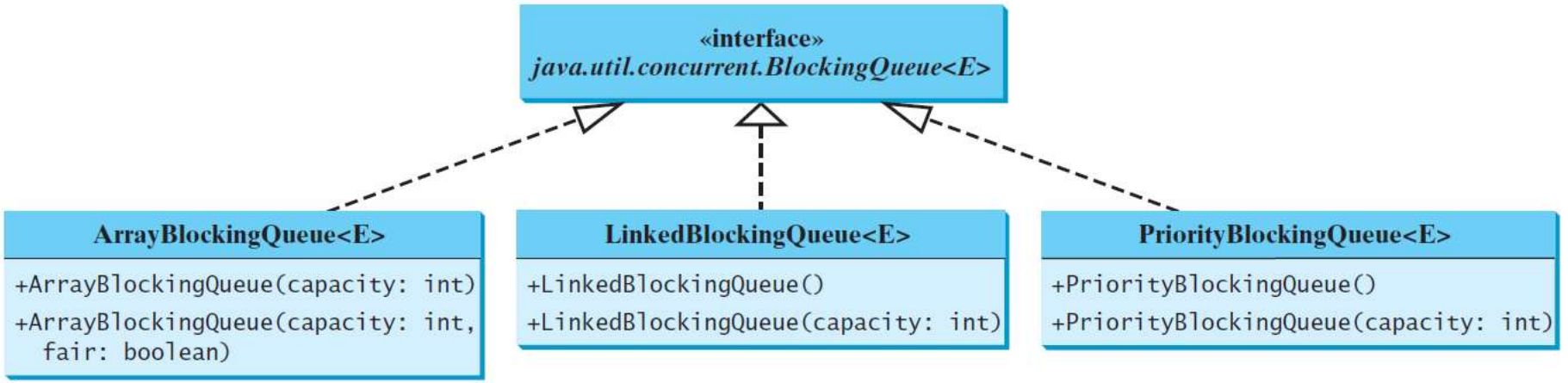
Blocking Queues (Optional)

§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.



Concrete Blocking Queues

Three concrete blocking queues ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are supported in JDK 1.5. 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.



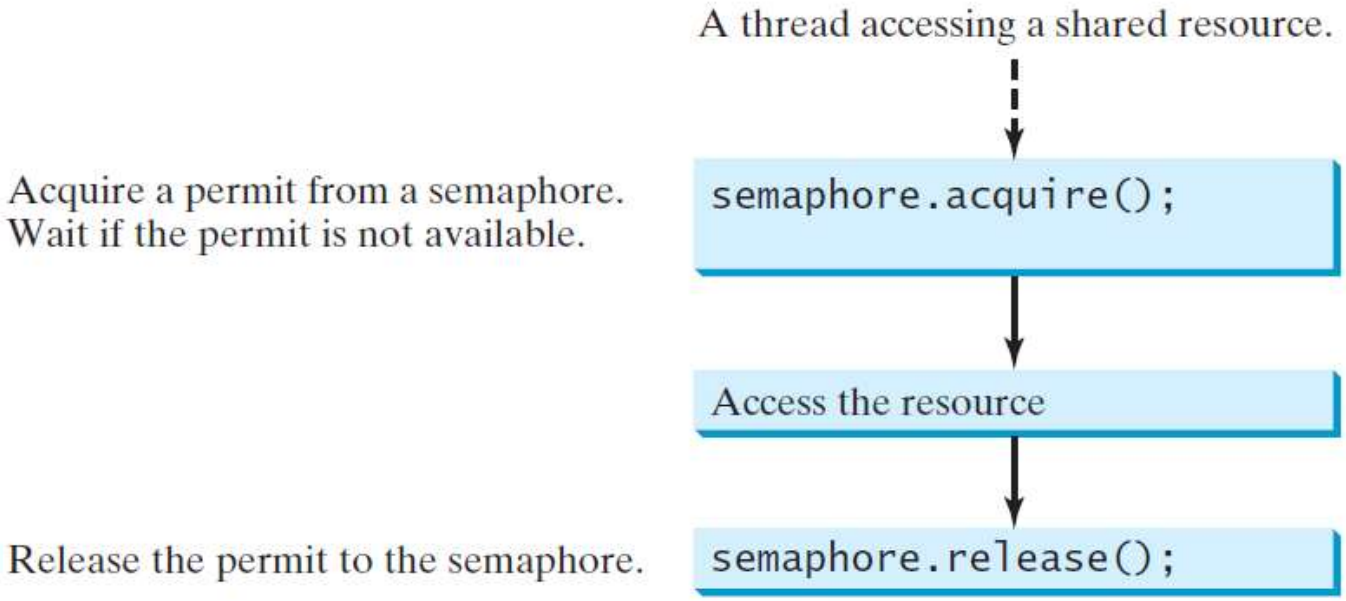
Producer/Consumer Using Blocking Queues

The program gives an example of using an ArrayBlockingQueue for the Consumer/Producer problem.

ConsumerProducerUsingBlockingQueue Run

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.



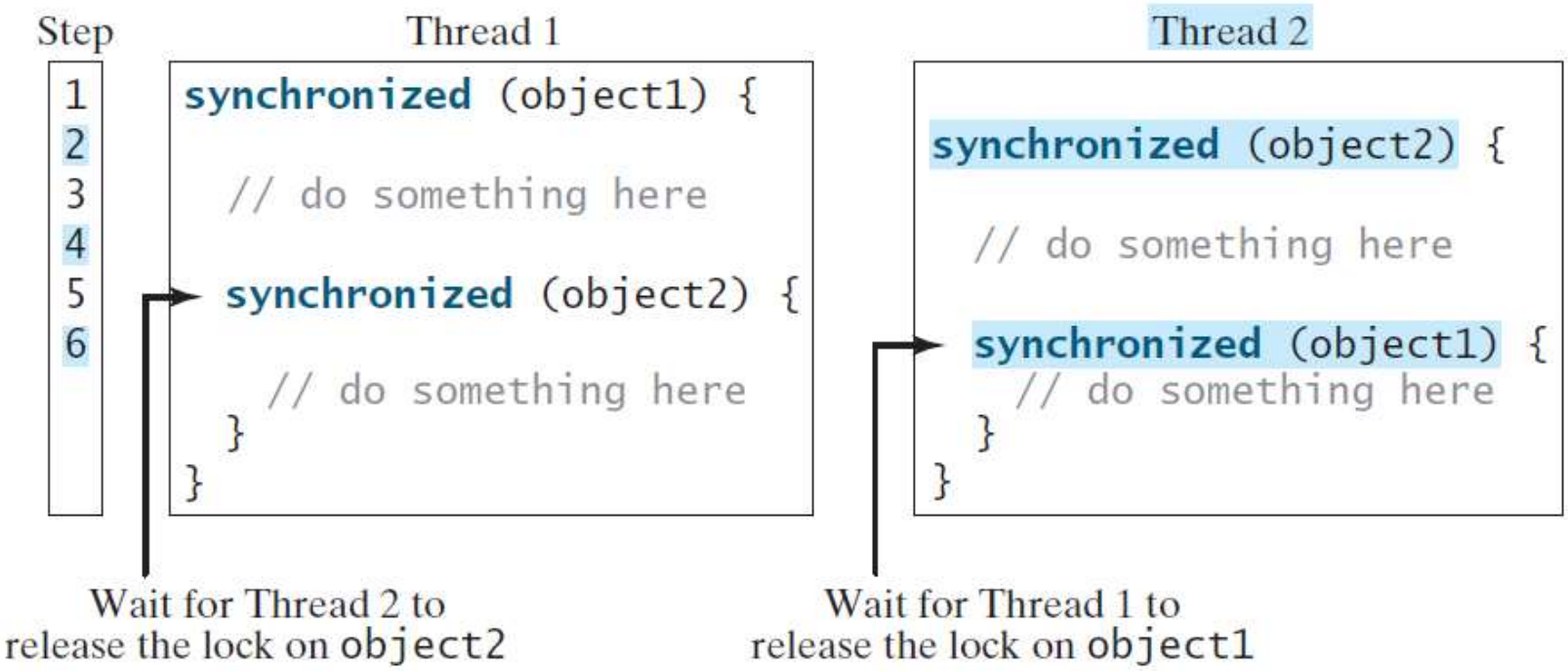
Creating Semaphores

To create a semaphore, you have to specify the number of permits with an optional fairness policy. A task acquires a permit by invoking the semaphore's acquire() method and releases the permit by invoking the semaphore's release() 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. 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.

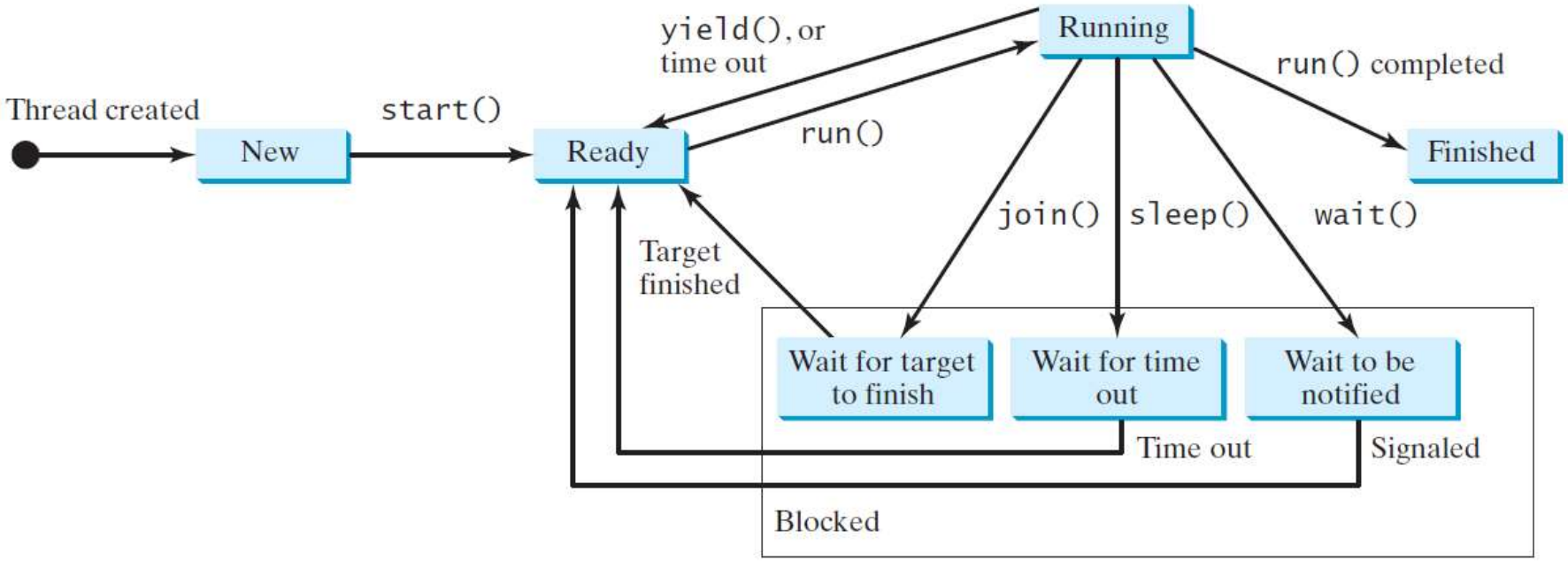


Preventing Deadlock

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, 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. The collections created using these methods are called *synchronization wrappers*.

Returns a synchronized collection.	
Returns a synchronized list from the specified list.	
Returns a synchronized map from the specified map.	
Returns a synchronized set from the specified set.	
Returns a synchronized sorted map from the specified sorted map.	
Returns a synchronized sorted set.	

The Fork/Join Framework

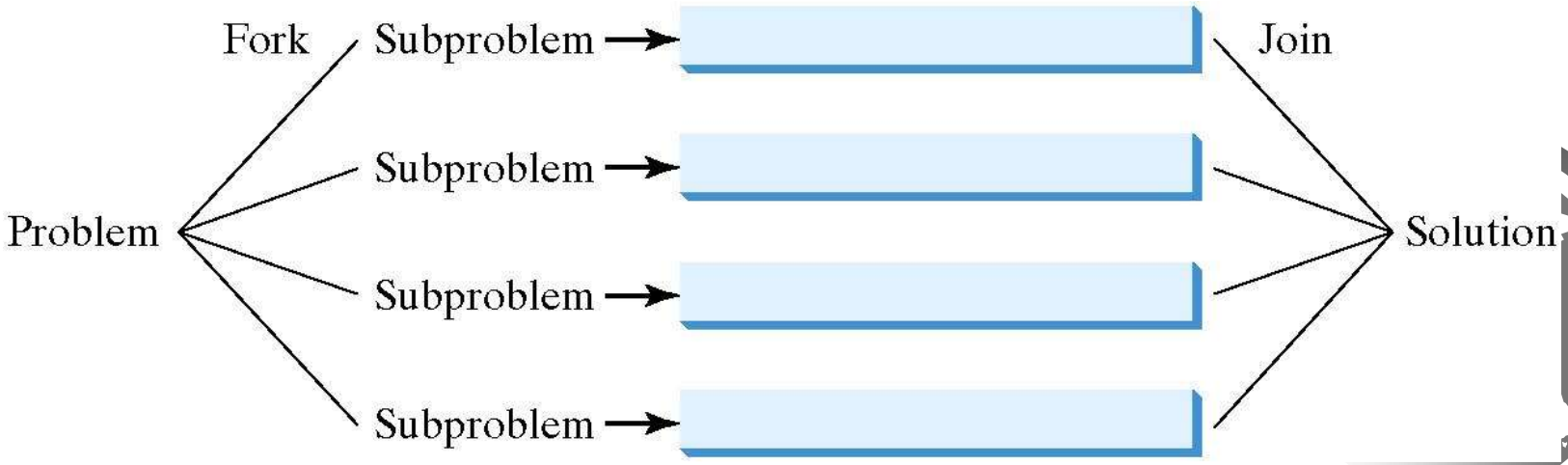
The widespread use of multicore systems has created a revolution in software. In order to benefit from multiple processors, software needs to run in parallel.

JDK 7 introduces the new Fork/Join Framework for parallel programming, which utilizes the multicore processors.

The Fork/Join Framework

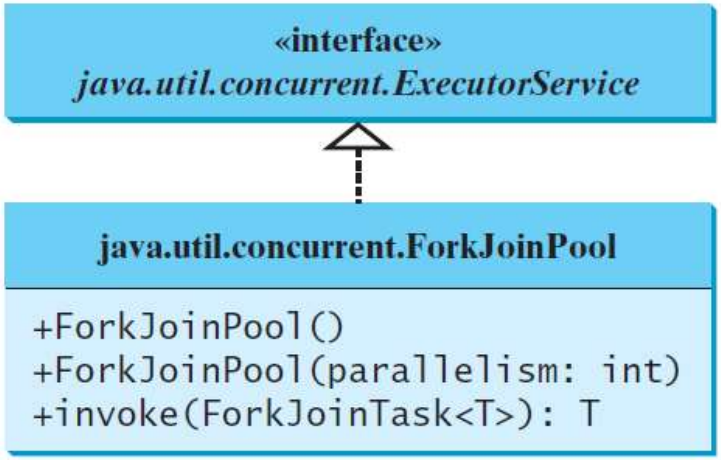
The Fork/Join Framework is used for parallel programming in Java.

In JDK 7's Fork/Join Framework, a *fork* can be viewed as an independent task that runs on a thread.



ForkJoinTask and ForkJoinPool

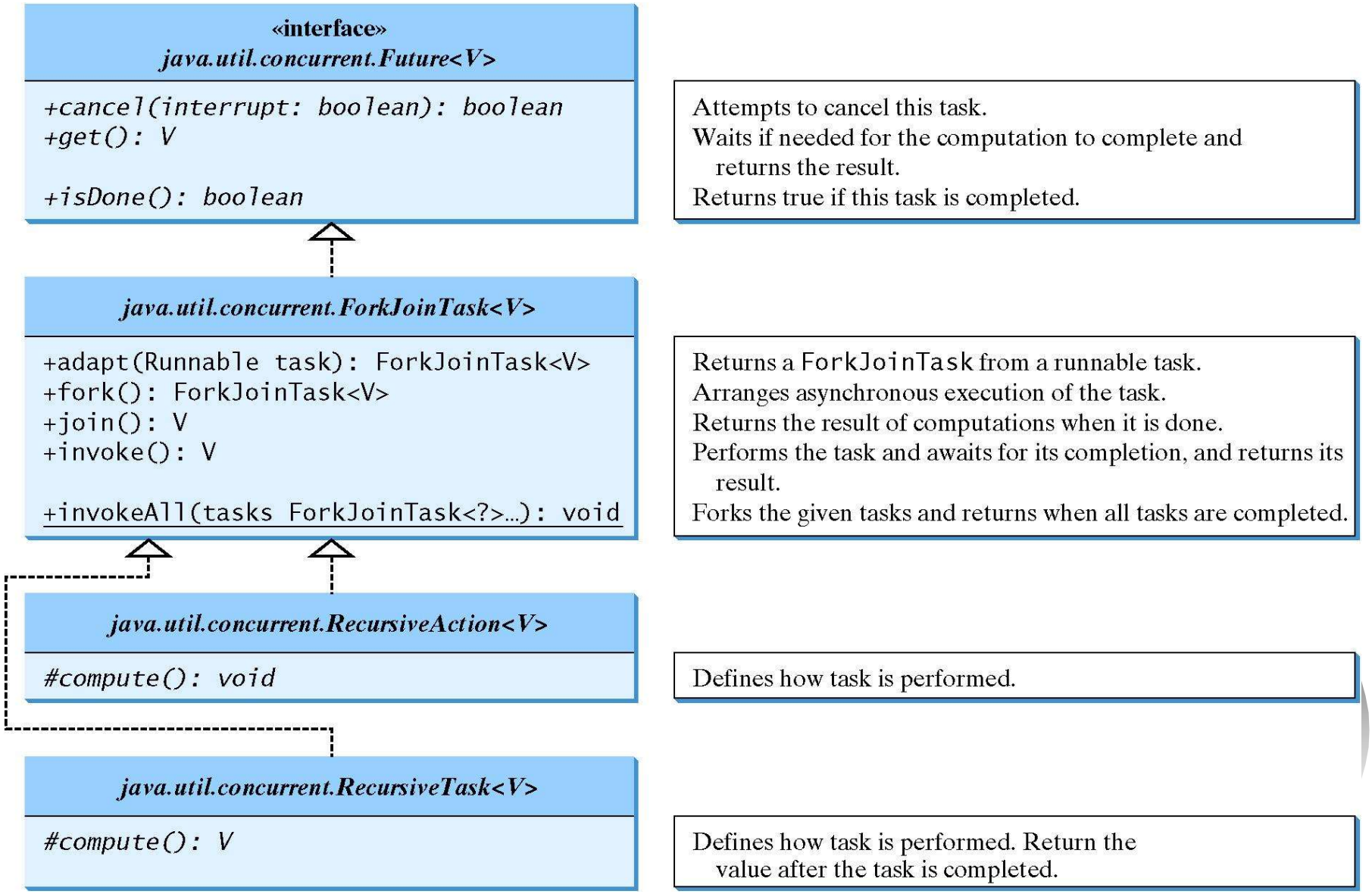
The framework defines a task using the **ForkJoinTask** class, and executes a task in an instance of **ForkJoinPool**.



See Figure 30.7

Creates a `ForkJoinPool` with all available processors.	
Creates a `ForkJoinPool` with the specified number of processors.	
Performs the task and returns its result upon completion.	

ForkJoinTask



Examples

ParallelMergeSort Run

ParallelMax Run

Companion Website

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.

Companion Website

Fail-Fast

The synchronization wrapper classes are thread-safe, but the iterator is *fail-fast*. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing java.util.ConcurrentModificationException, which is a subclass of RuntimeException. To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it. For example, suppose you want to traverse a set, you have to write the code like this:

```
Set hashSet = Collections.synchronizedSet(new HashSet());
synchronized (hashSet) { // Must synchronize it
  Iterator iterator = hashSet.iterator();
  while (iterator.hasNext()) {
    System.out.println(iterator.next());
  }
}
```

Failure to do so may result in nondeterministic behavior, such as ConcurrentModificationException.