Chapter 30 Multithreading and Networking

Threads Concept

Multiple threads on multiple CPUs

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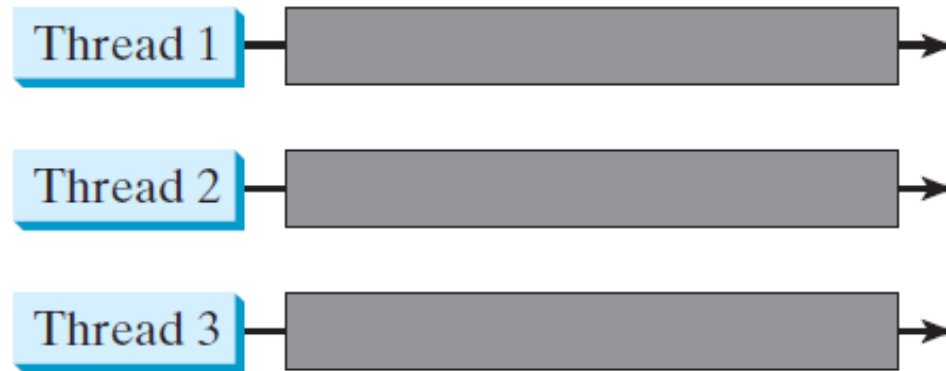
Multiple threads sharing a single CPU

Thread 1	
Thread 2	
Thread 3	
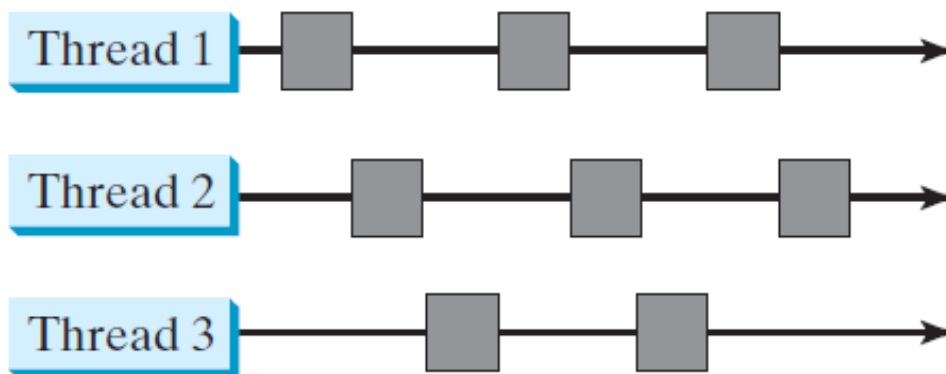
Creating Tasks and Threads

```
java.lang.Runnable  <---------  TaskClass
```

```
// 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
    ...
  }
  ...
}
```

(a)

```
// 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();
    ...
  }
  ...
}
```
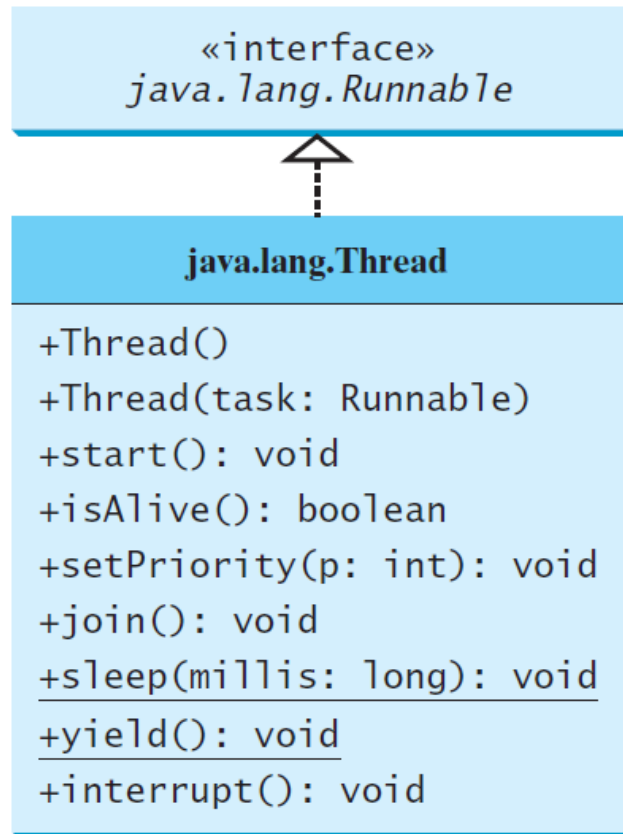
(b)

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

The Thread Class

«interface»
java.lang.Runnable

```
           java.lang.Thread

+Thread()
+Thread(task: Runnable)
+start(): void
+isAlive(): boolean
+setPriority(p: int): void
+join(): void
+sleep(millis: long): void
+yield(): void
+interrupt(): void
```

Creates an empty thread.

Creates a thread for a specified task.

Starts the thread that causes the run() method to be invoked by the JVM.

Tests whether the thread is currently running.

Sets priority p (ranging from 1 to 10) for this thread.

Waits for this thread to finish.

Puts a thread to sleep for a specified time in milliseconds.

Causes a thread to pause temporarily and allow other threads to execute.

Interrupts this thread.

ExtendedThread

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 TaskThreadDemo.java as follows:

```java
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 just once then the characters follows.

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 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
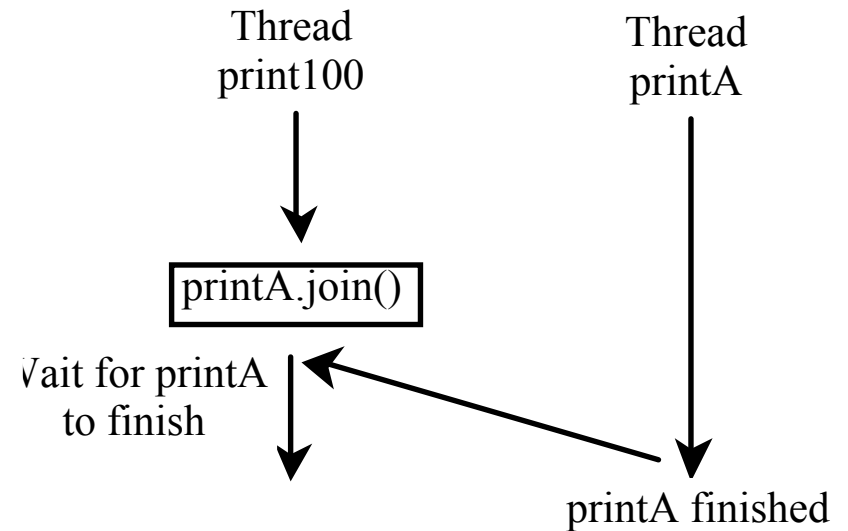
printA finished

The numbers after 50 are printed after thread printC 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.

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
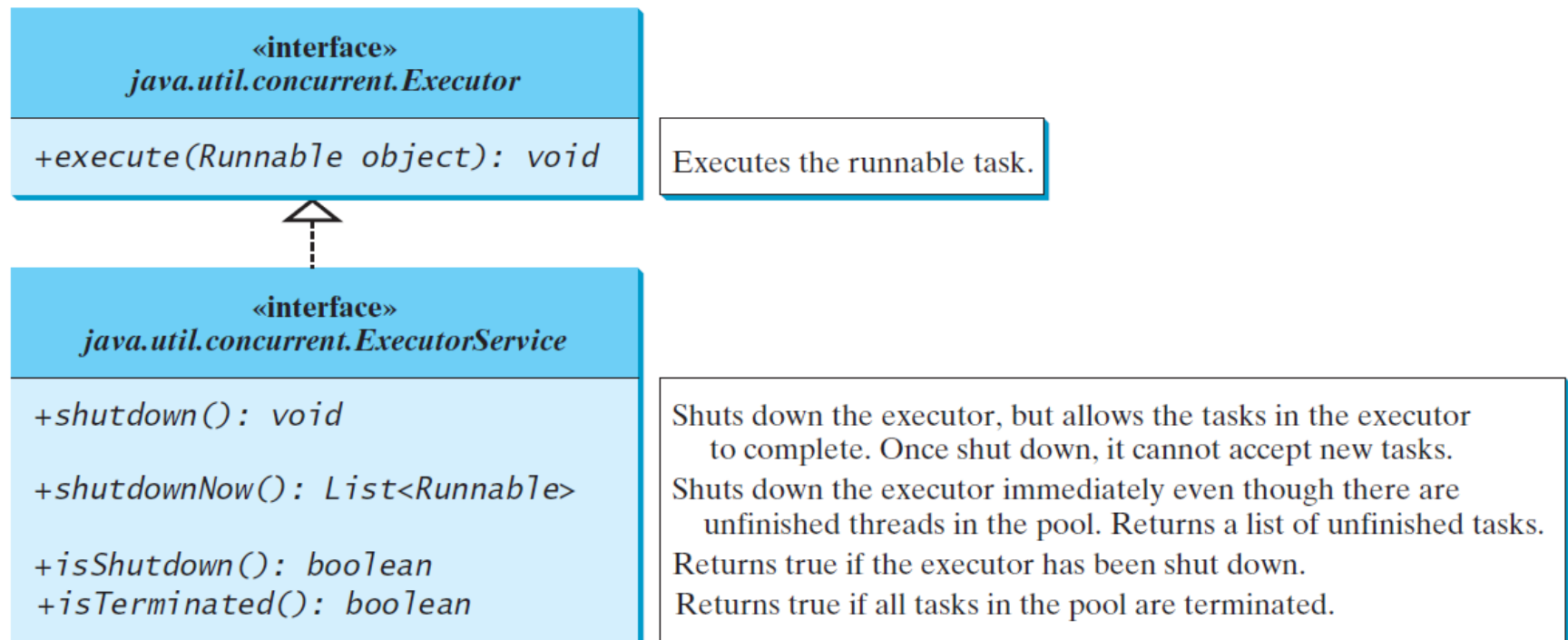`Thread.MIN_PRIORITY`
`Thread.MAX_PRIORITY`
`Thread.NORM_PRIORITY`

Example: Flashing Text

FlashText _ □ ✕	FlashText _ □ ✕	FlashText _ □ ✕
Welcome		Welcome
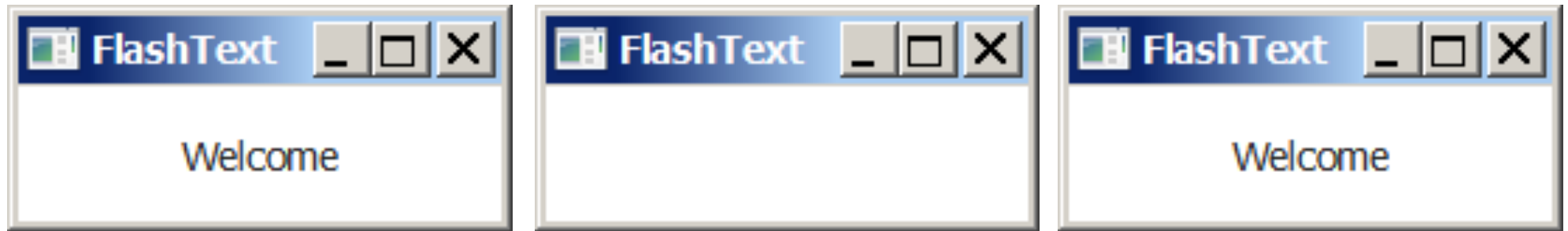
FlashText

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

+shutdownNow(): List<Runnable>

+isShutdown(): boolean
+isTerminated(): boolean
```

Shuts down the executor, but allows the tasks in the executor to complete. Once shut down, it cannot accept new tasks.
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.
Returns true if the executor has been shut down.
Returns true if all tasks in the pool are terminated.

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

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 =balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = 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.
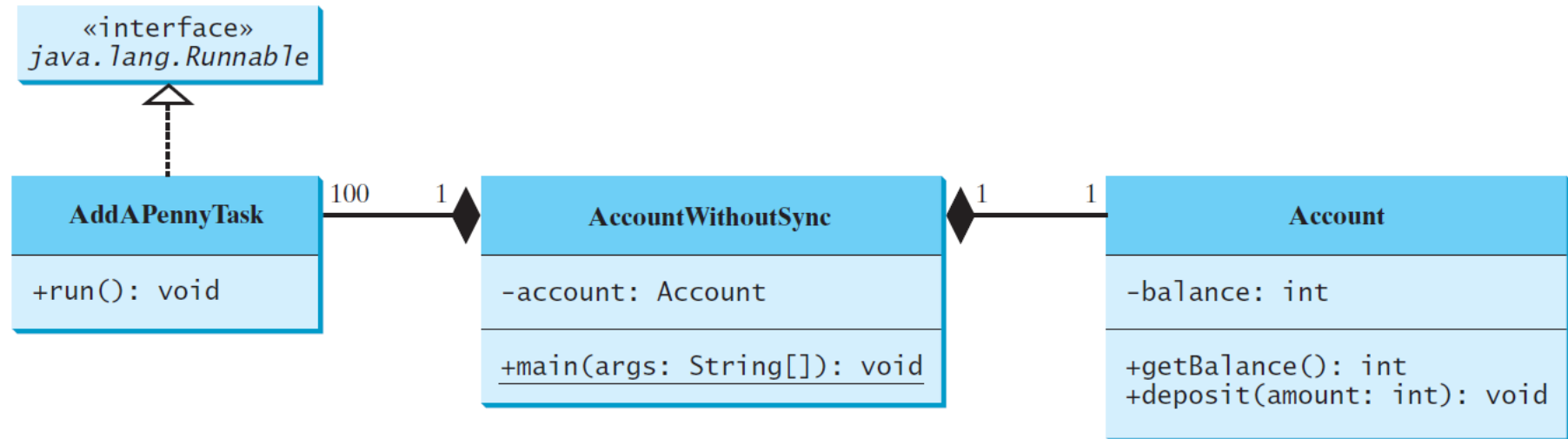
AccountWithoutSync

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 *AccountWithoutSync.java 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 *AccountWithoutSync.java is*, 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)

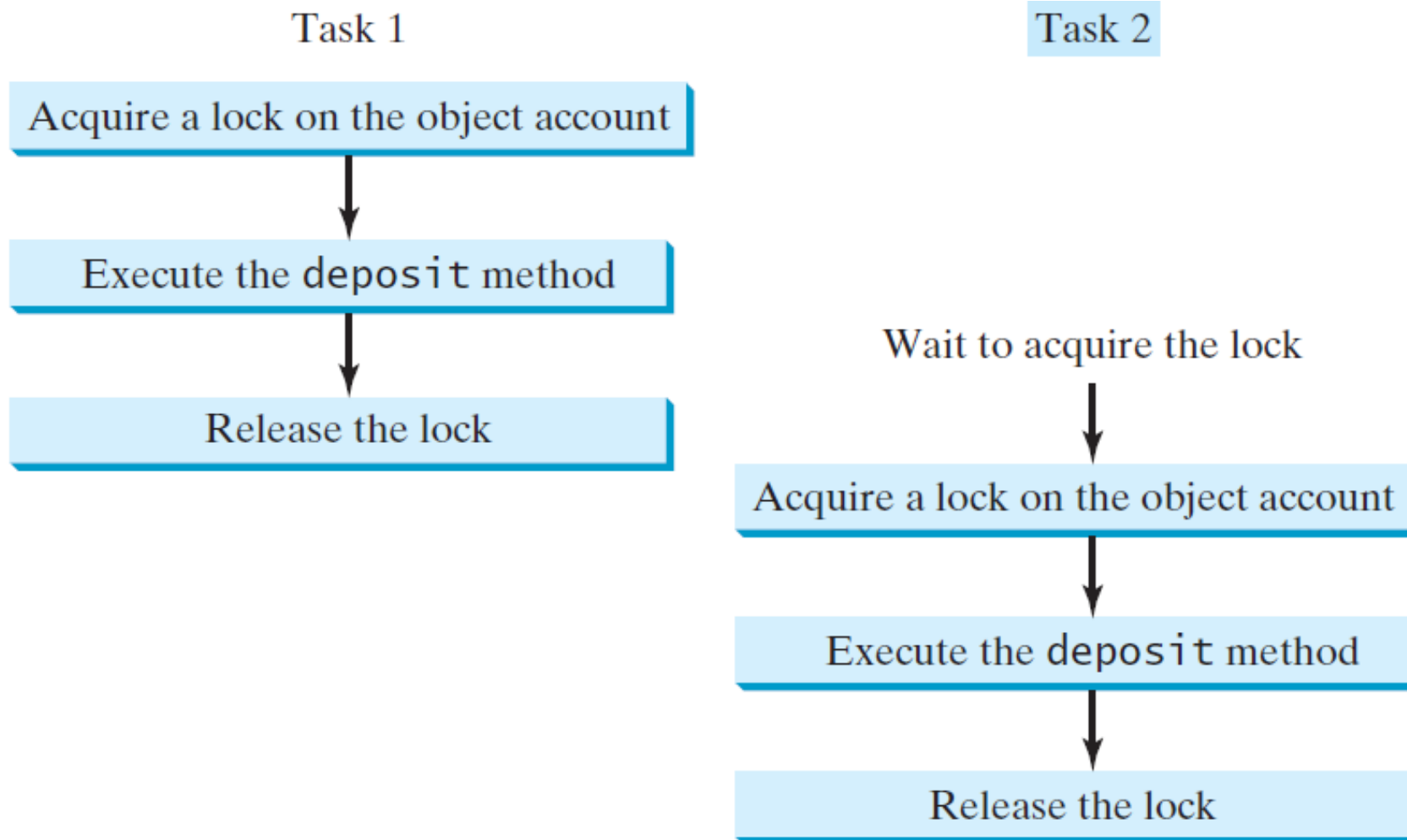# Synchronizing Instance Methods and Static Methods

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

Task 1

Acquire a lock on the object account

↓

Execute the `deposit` method

↓

Release the lock

Task 2

Wait to acquire the lock

↓

Acquire a lock on the object account

↓

Execute the `deposit` method

↓

Release the lock

20

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
}
```
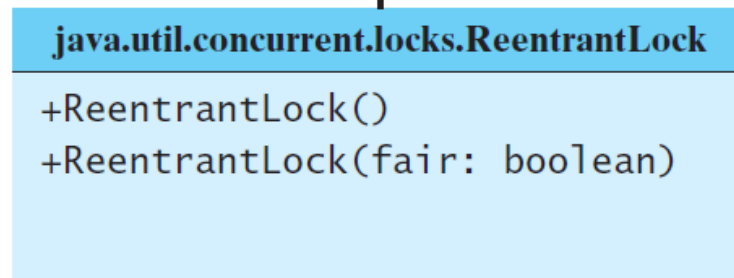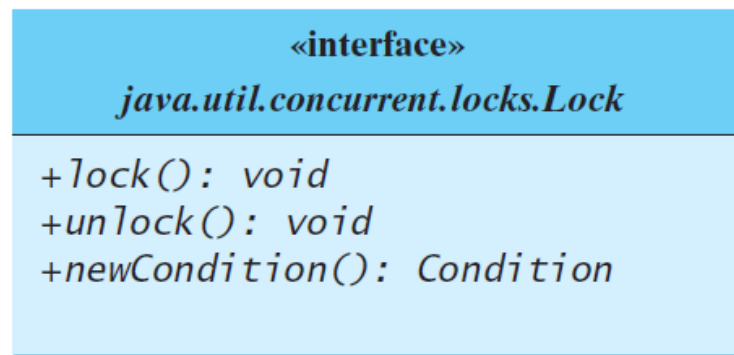
This method is equivalent to

```
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.

```
«interface»
java.util.concurrent.locks.Lock

+lock(): void
+unlock(): void
+newCondition(): Condition
```

Acquires the lock.
Releases the lock.
Returns a new Condition instance that is bound to this Lock instance.

AccountWithSyncUsingLock

```
java.util.concurrent.locks.ReentrantLock

+ReentrantLock()
+ReentrantLock(fair: boolean)
```

Same as ReentrantLock(false).
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.

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.

«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.
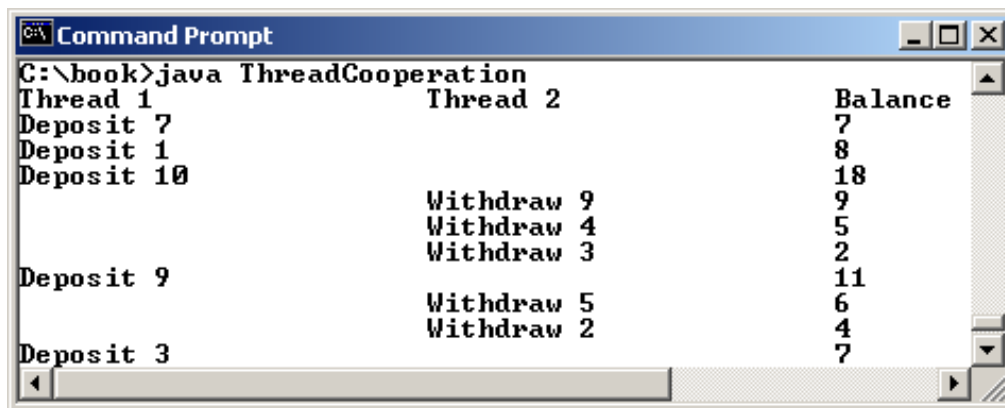
Cooperation Among Threads

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.

Withdraw Task

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

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

```
balance -= withdrawAmount
```

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

Deposit Task

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

```
balance += depositAmount
```

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

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

25
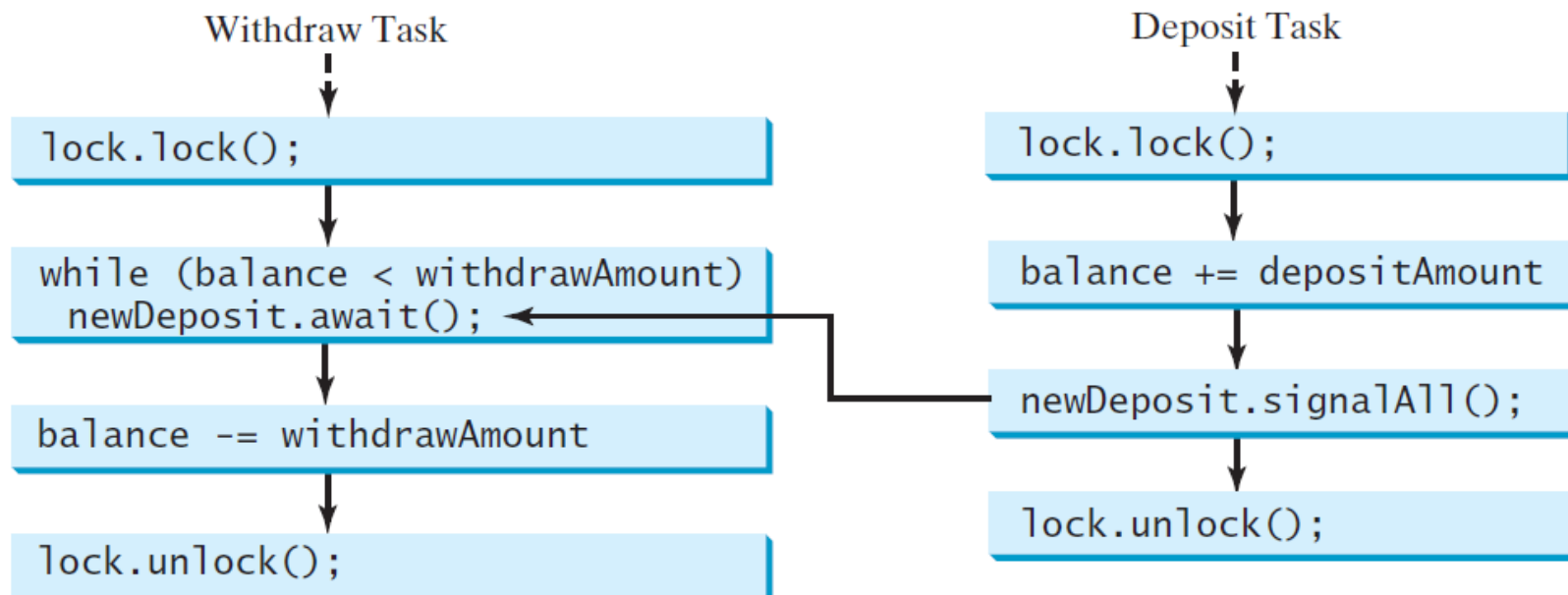
Example: Thread Cooperation

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

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

```
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();
  }
}
```
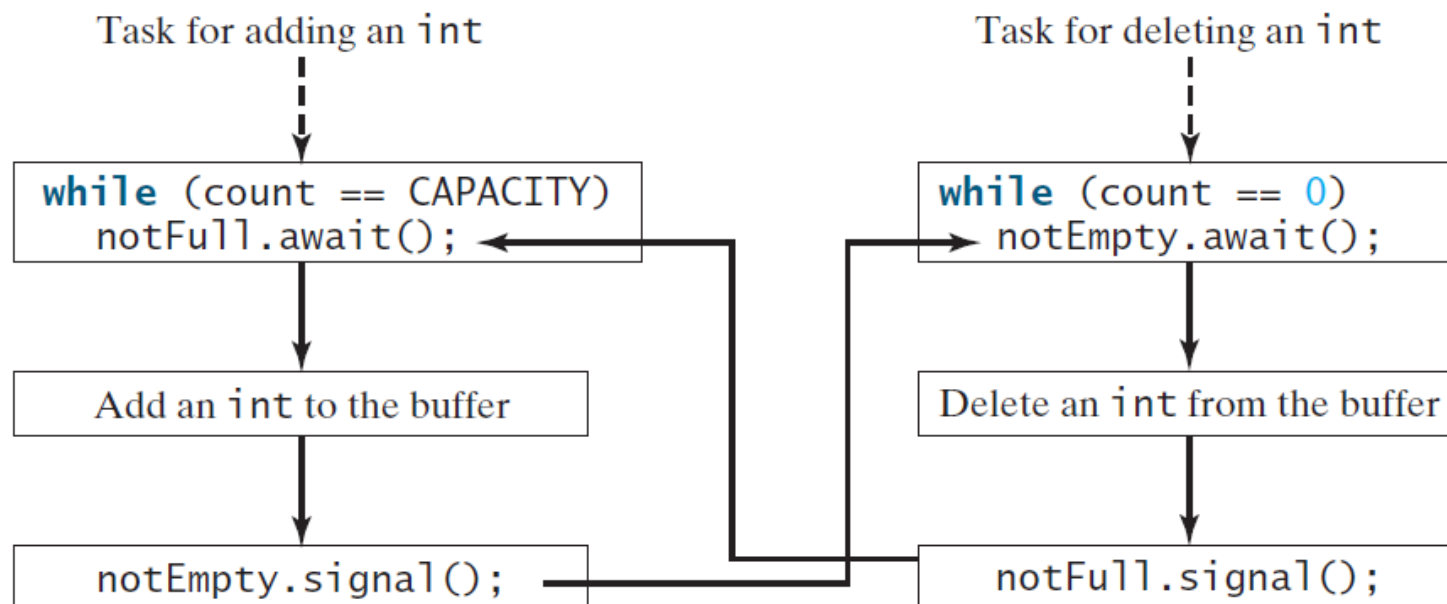
```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or anObject.notifyAll();
  ...
}
```

✦ 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.

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` from the buffer

```
notEmpty.signal();
```

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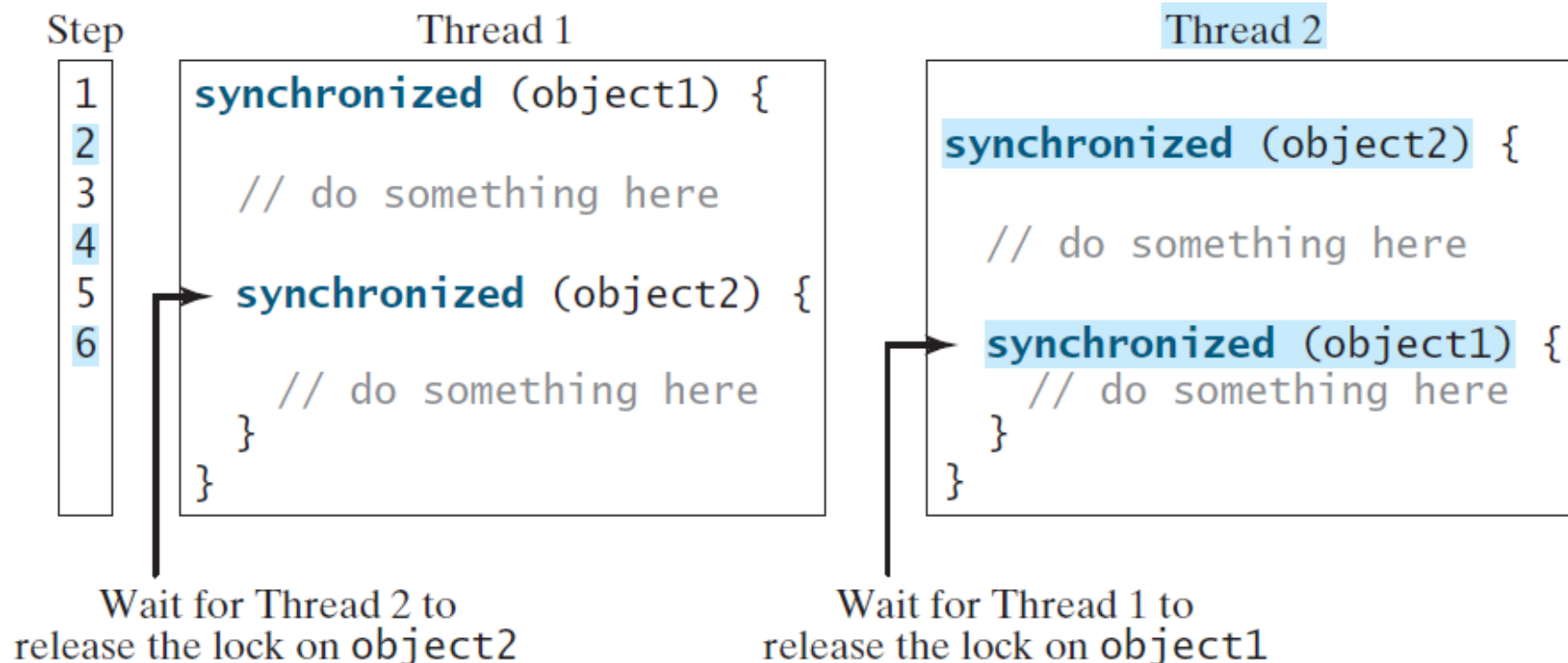
Case Study: Producer/Consumer (Optional)

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

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.

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, 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.
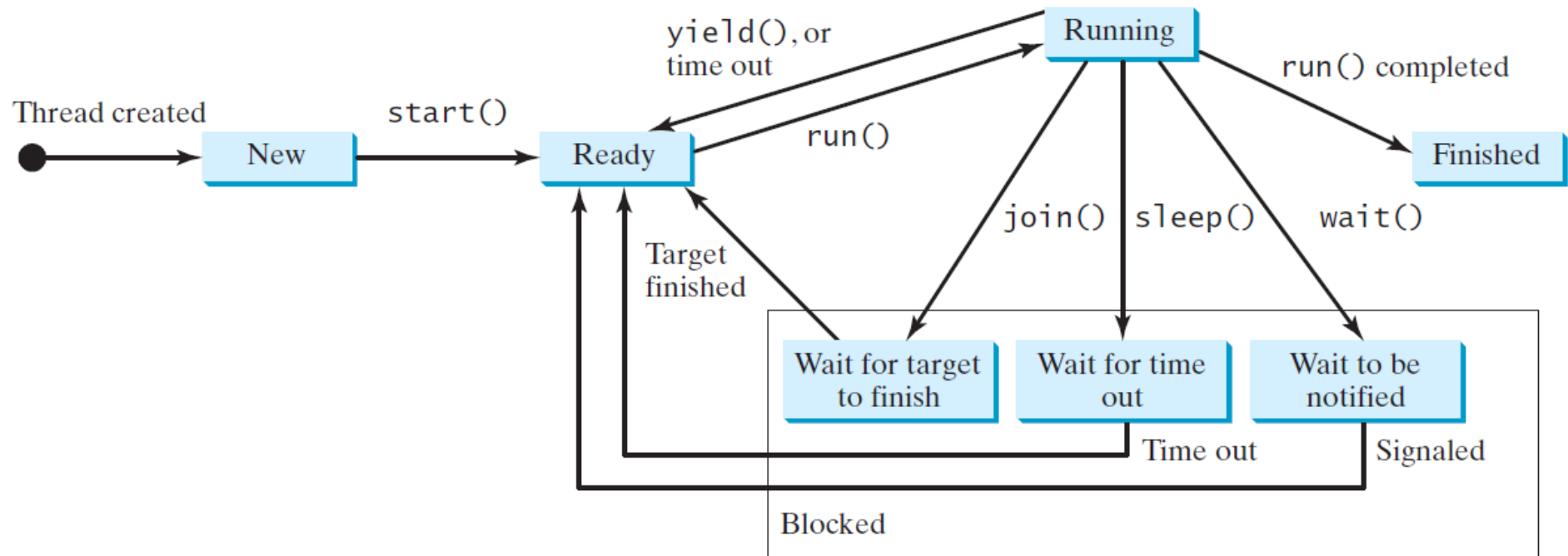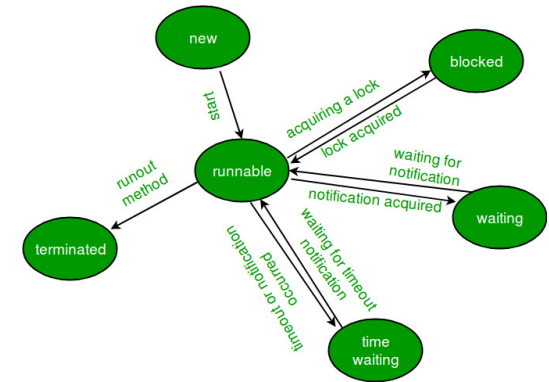
Synchronized Collections

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*.

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.

Stream Socket vs. Datagram Socket

✦ A dedicated point-to-point channel between a client and server.

✦ Use TCP (Transmission Control Protocol) for data transmission.

✦ Lossless and reliable.

✦ Sent and received in the same order.
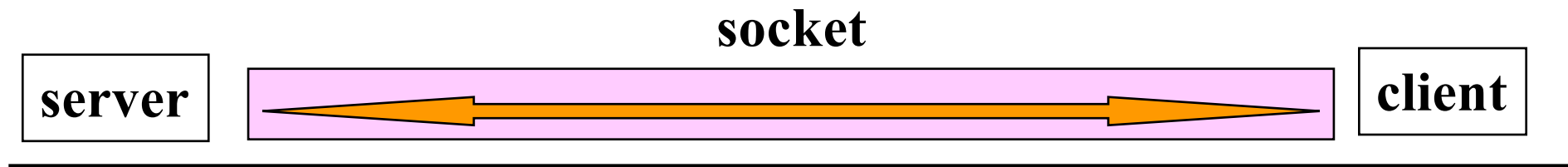
socket

server		client
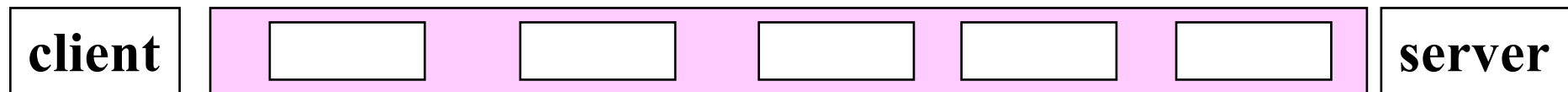
datagram

client		server

✦ No dedicated point-to-point channel between a client and server.

✦ Use UDP (User Datagram Protocol) for data transmission.

Datagram
socket

✦ May lose data and not 100% reliable.

✦ Data may not received in the same order as sent.

Socket based Service

Network Service

IP,port

Socket

Client

20   ftp

21   telnet

23   Mail

finger

端口

I/O buffer

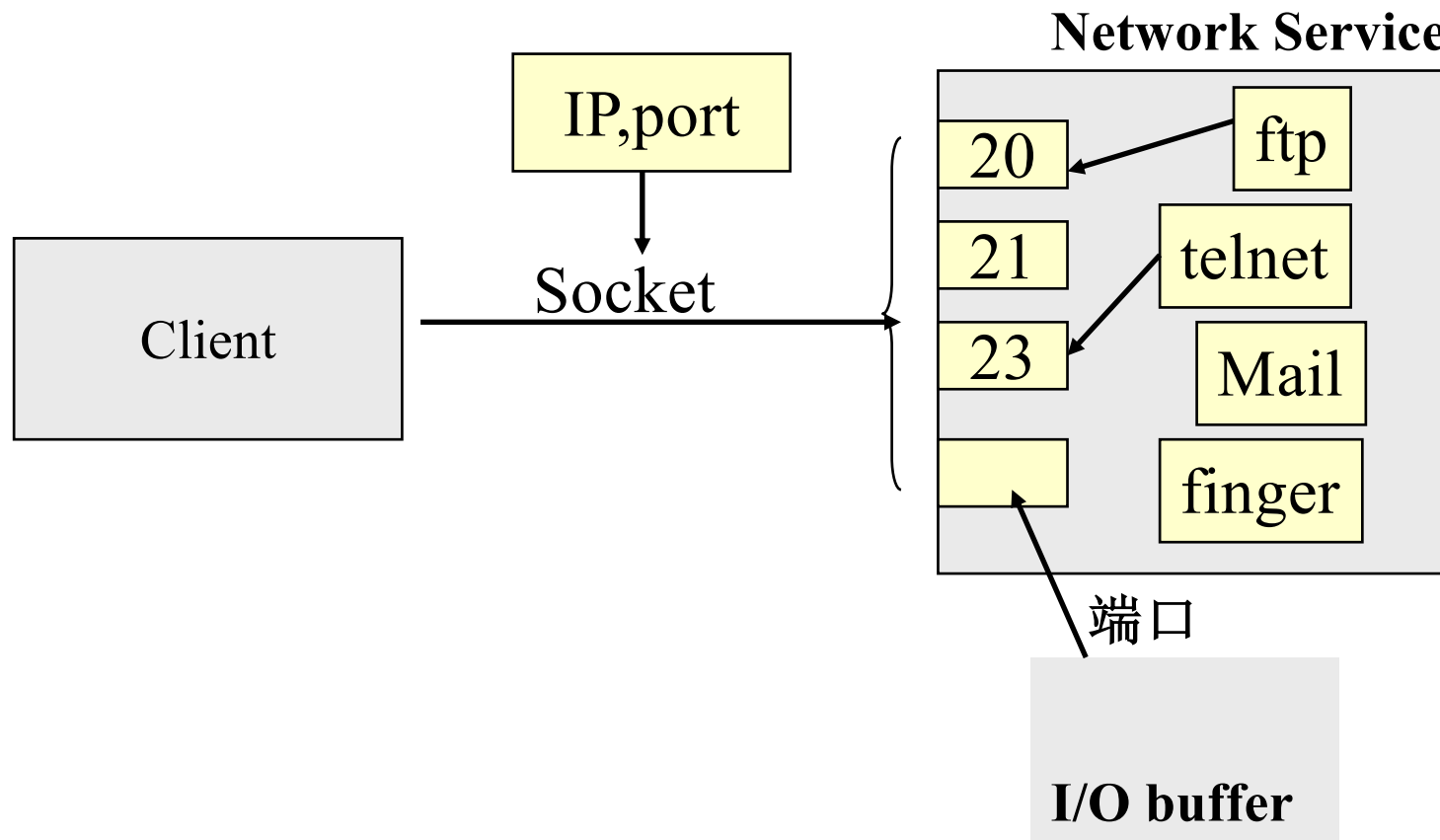# Client/Server Communications

The server must be running when a client starts. The server waits for a connection request from a client. To establish a server, you need to create a server socket and attach it to a port, which is where the server listens for connections.

After the server accepts the connection, communication between server and client is conducted the same as for I/O streams.

After a server socket is created, the server can use this statement to listen for connections.
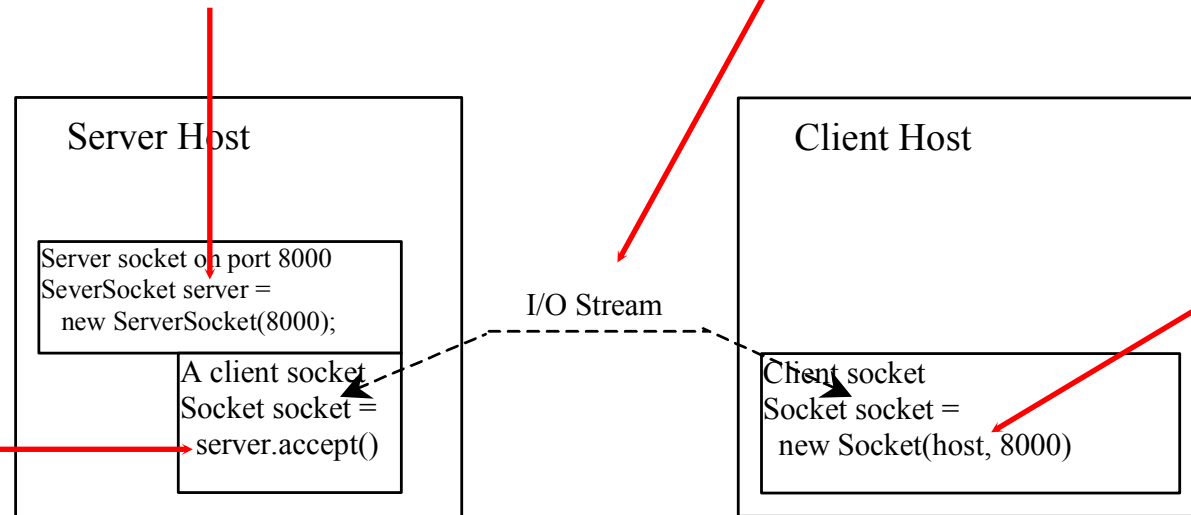
The client issues this statement to request a connection to a server.

Server Host

Client Host

Server socket on port 8000
SeverSocket server =
  new ServerSocket(8000);

I/O Stream

A client socket
Socket socket =
  server.accept()

Client socket
Socket socket =
  new Socket(host, 8000)

Data Transmission through Sockets

Server		Client

```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server = new ServerSocket(port);
socket = server.accept();
in = new DataInputStream
   (socket.getInputStream());
out = new DataOutStream
   (socket.getOutputStream());
System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

Connection Request

I/O Streams

```
int port = 8000;
String host = "localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;


socket = new Socket(host, port);
in = new DataInputStream
   (socket.getInputStream());
out = new DataOutputStream
   (socket.getOutputStream());
out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

InputStream input = socket.getInputStream();

OutputStream output = socket.getOutputStream();

A Client/Server Example

✦ Problem: Write a client to send data to a server. The server receives the data, uses it to produce a result, and then sends the result back to the client. The client displays the result on the console.

compute area

String

Server	←	Client

String

Server Code

Client Code
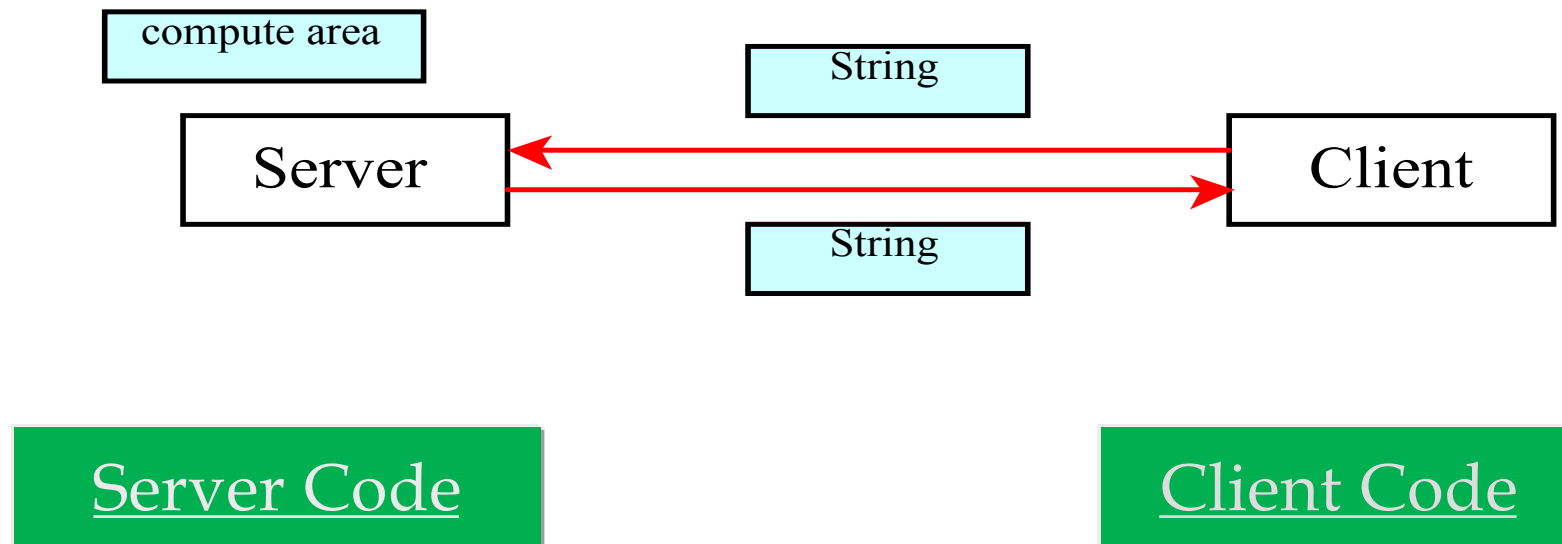
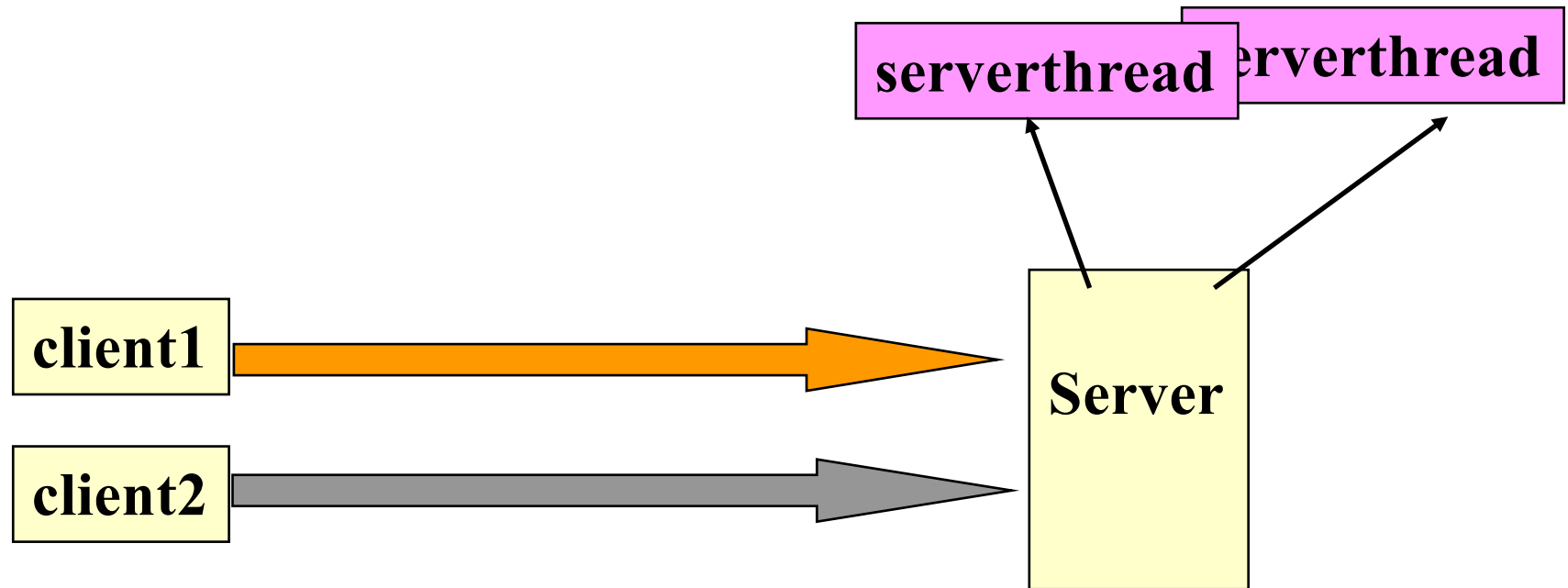Note: Start the server, then the client.

The <u>InetAddress</u> Class

Occasionally, you would like to know who is connecting to the server. You can use the InetAddress class to find the client's host name and IP address. The InetAddress class models an IP address. You can use the statement shown below to create an instance of InetAddress for the client on a socket.

```
InetAddress inetAddress = socket.getInetAddress();
```
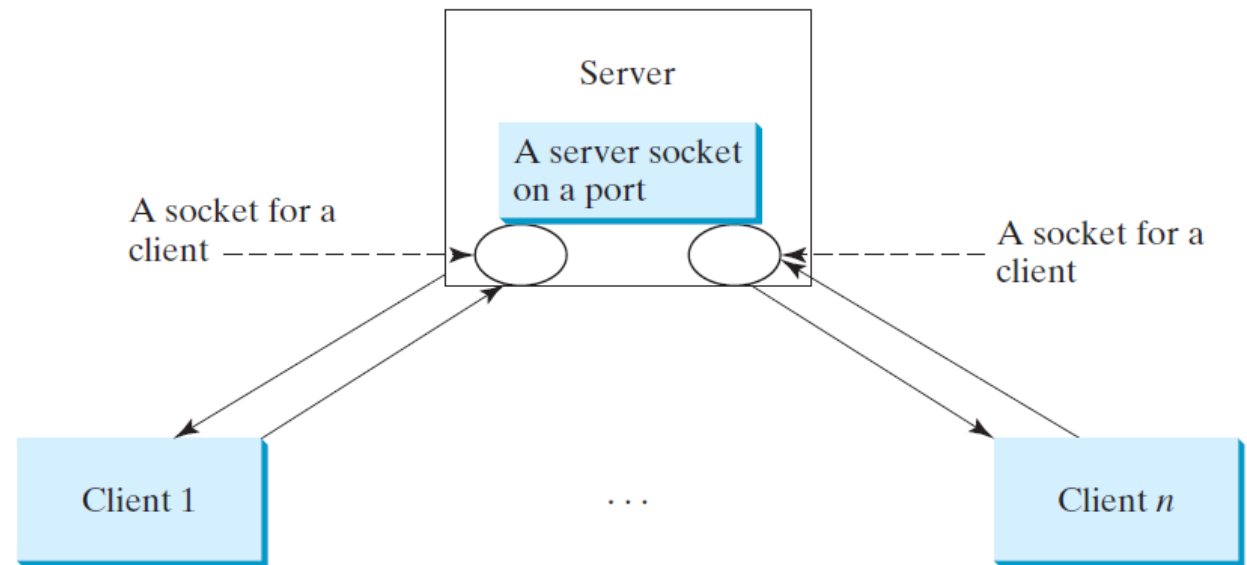
Next, you can display the client's host name and IP address, as follows:

```
System.out.println("Client's host name is " +
    inetAddress.getHostName());
System.out.println("Client's IP Address is " +
    inetAddress.getHostAddress());
```

Serving Multiple Clients
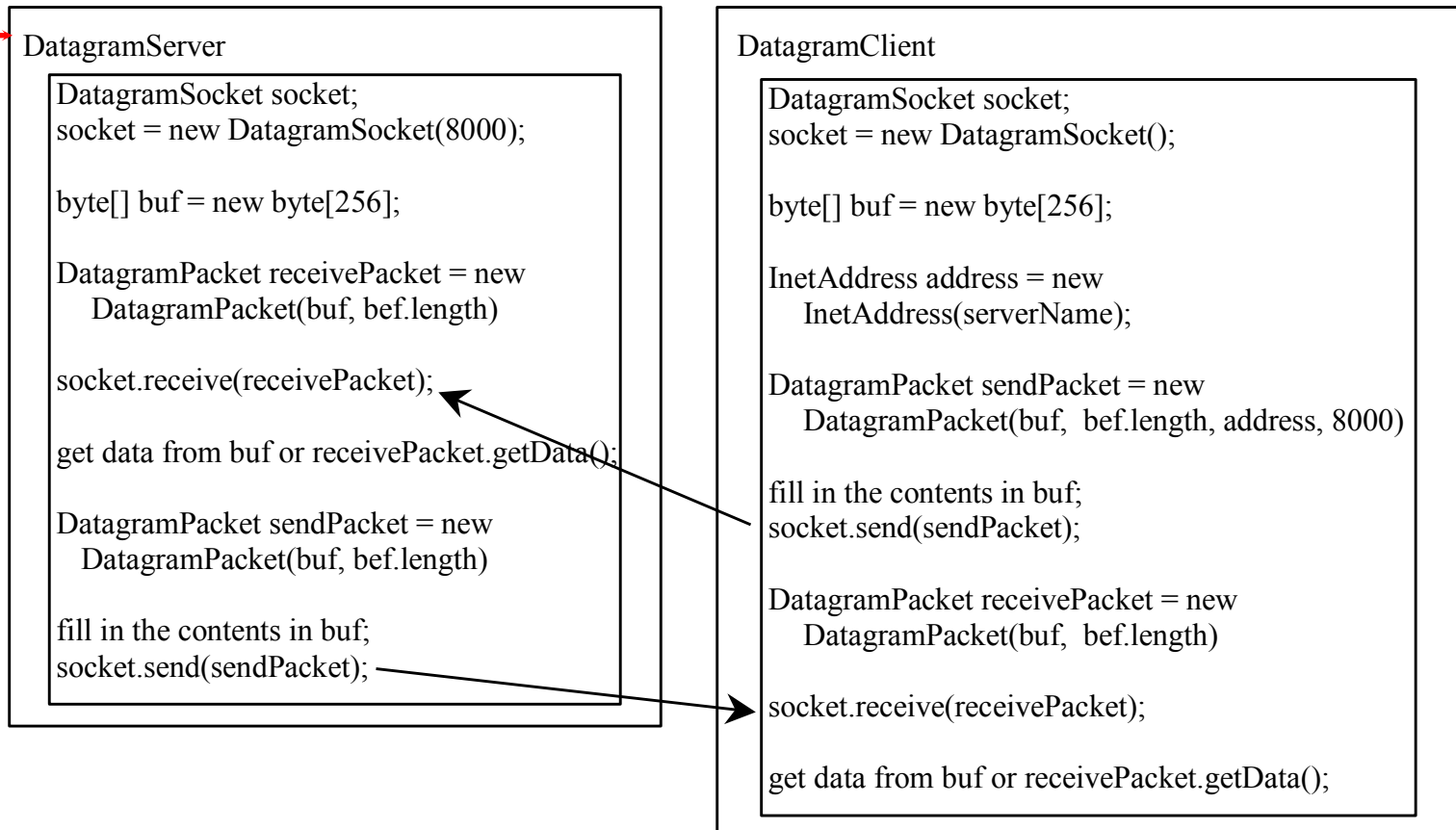
Example: Serving Multiple Clients

ServerWithThreadPool

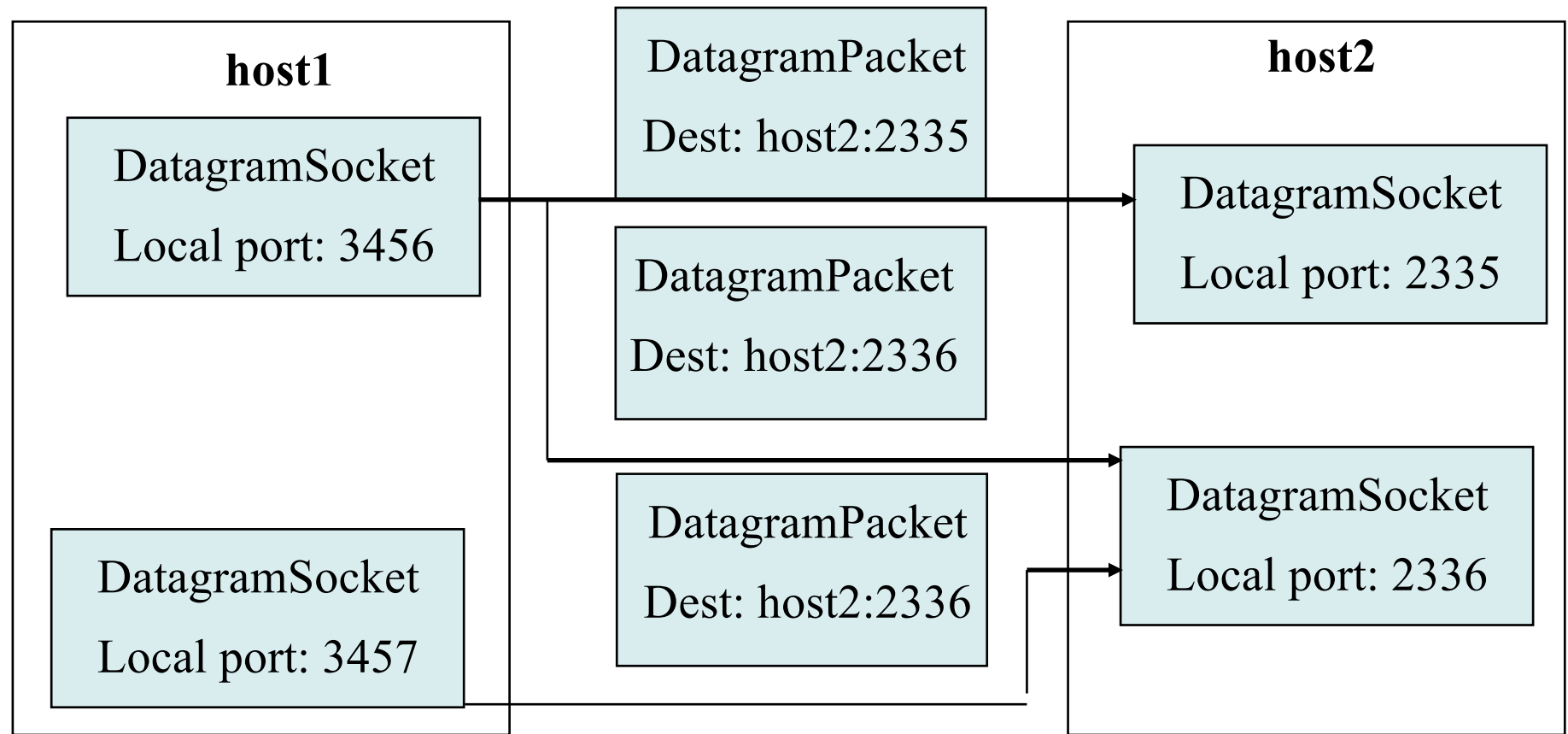Note: Start the server first, then start multiple clients.

Datagram Programming

Datagram programming is different from stream socket programming in the sense that there is no concept of a ServerSocket for datagrams. Both client and server use DatagramSocket to send and receive packets.

Designate one a server

DatagramServer

```
DatagramSocket socket;
socket = new DatagramSocket(8000);

byte[] buf = new byte[256];

DatagramPacket receivePacket = new
    DatagramPacket(buf, bef.length)

socket.receive(receivePacket);

get data from buf or receivePacket.getData();

DatagramPacket sendPacket = new
    DatagramPacket(buf, bef.length)

fill in the contents in buf;
socket.send(sendPacket);
```

DatagramClient

```
DatagramSocket socket;
socket = new DatagramSocket();

byte[] buf = new byte[256];

InetAddress address = new
    InetAddress(serverName);

DatagramPacket sendPacket = new
    DatagramPacket(buf,  bef.length, address, 8000)

fill in the contents in buf;
socket.send(sendPacket);

DatagramPacket receivePacket = new
    DatagramPacket(buf,  bef.length)

socket.receive(receivePacket);

get data from buf or receivePacket.getData();
```

*DatagramSocket DatagramPacket

host1

DatagramSocket
Local port: 3456

DatagramSocket
Local port: 3457

DatagramPacket
Dest: host2:2335

DatagramPacket
Dest: host2:2336

DatagramPacket
Dest: host2:2336

host2

DatagramSocket
Local port: 2335

DatagramSocket
Local port: 2336

ServerWithUDP

ClientWithUDP

Note: Start the server first, then start multiple clients.