Object-Based Systems Programming Chapter 30: Multithreading and Parallel Programming

Dr. Amjad Hawash

30.1 Introduction

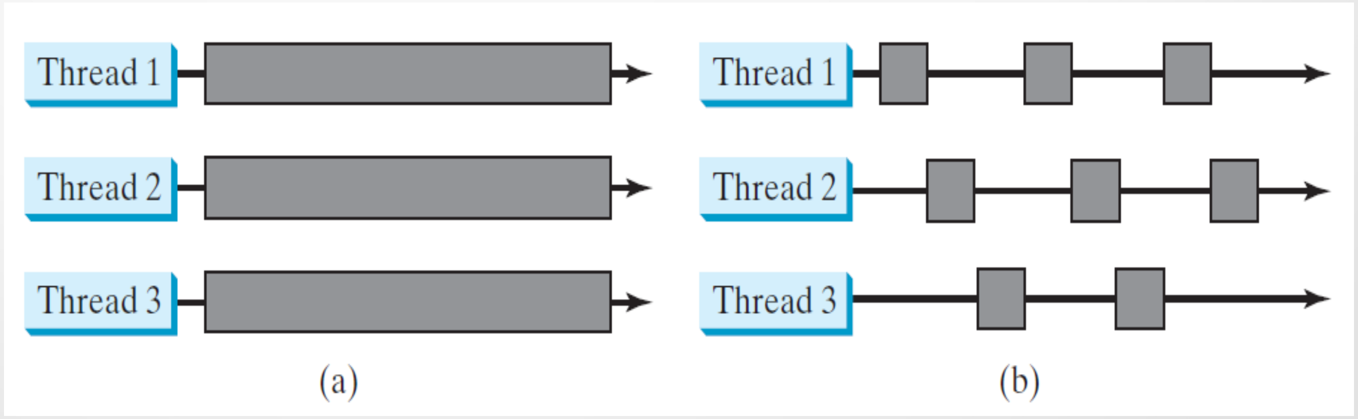
- Multithreading enables multiple tasks in a program to be executed concurrently.
- In many programming languages, you have to invoke system-dependent procedures and functions to implement multithreading.

30.2 Thread Concepts

- A program may consist of many tasks that can run concurrently.
- A thread is the flow of execution, from beginning to end, of a task.
- With Java, you can launch multiple threads from a program concurrently.
- These threads can be executed simultaneously in multiprocessor systems.

30.2 Thread Concepts

- In single-processor systems, the multiple threads share CPU time, known as time sharing, and the operating system is responsible for scheduling and allocating resources to them.
- This arrangement is practical because most of the time the CPU is idle.
- It does nothing, for example, while waiting for the user to enter data.



30.2 Thread Concepts

- Multithreading can make your program more responsive and interactive, as well as enhance performance.
- For example, a good word processor lets you print or save a file while you are typing.
- In some cases, multithreaded programs run faster than single-threaded programs even on single-processor systems.

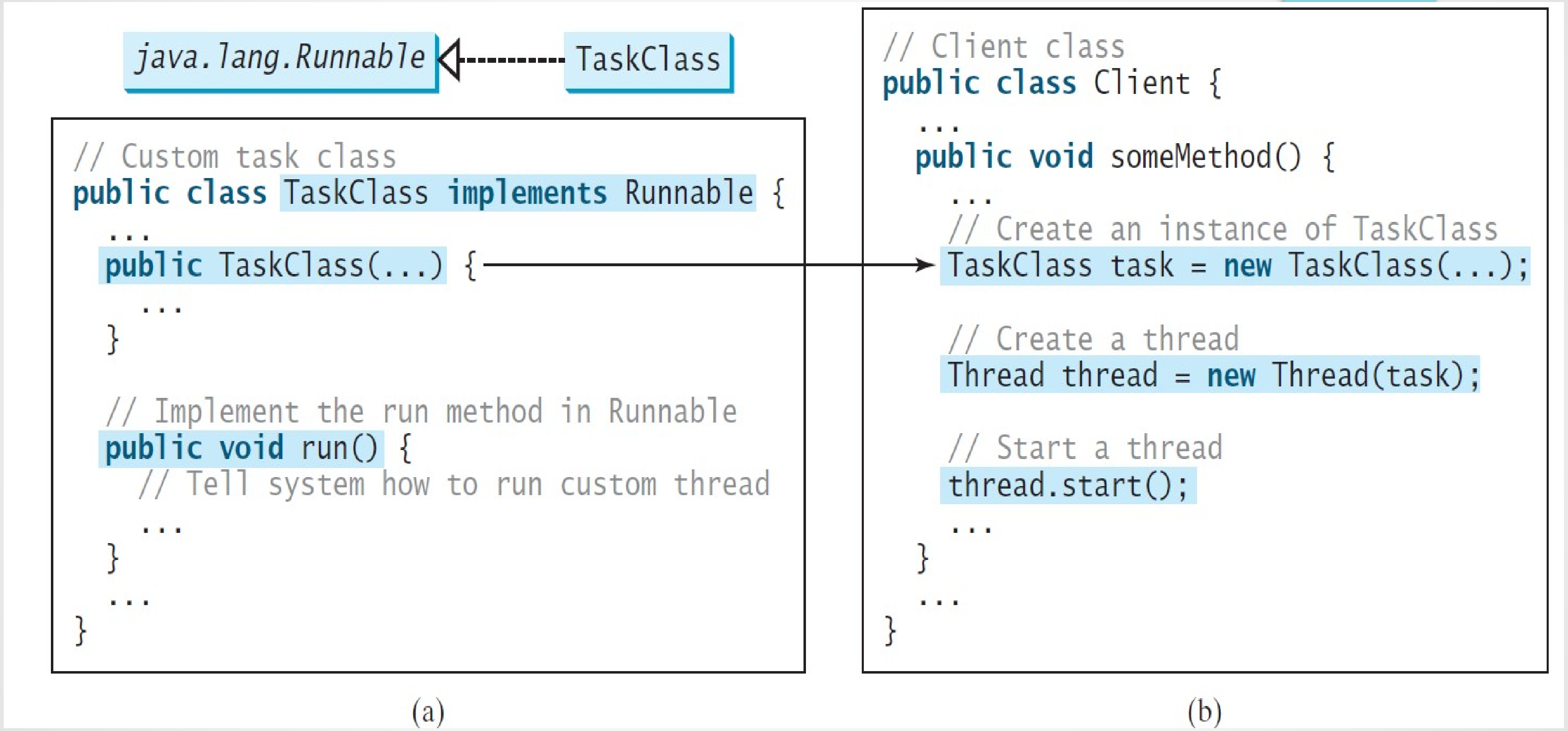
30.2 Thread Concepts

- In Java, each task is an instance of the Runnable interface, also called a runnable object.
- A thread is essentially an object that facilitates the execution of a task.

30.3 Creating Tasks and Threads

- A task class must implement the Runnable interface.
- Tasks are objects.
- To create tasks, you have to first define a class for tasks, which implements the Runnable interface.

30.3 Creating Tasks and Threads



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

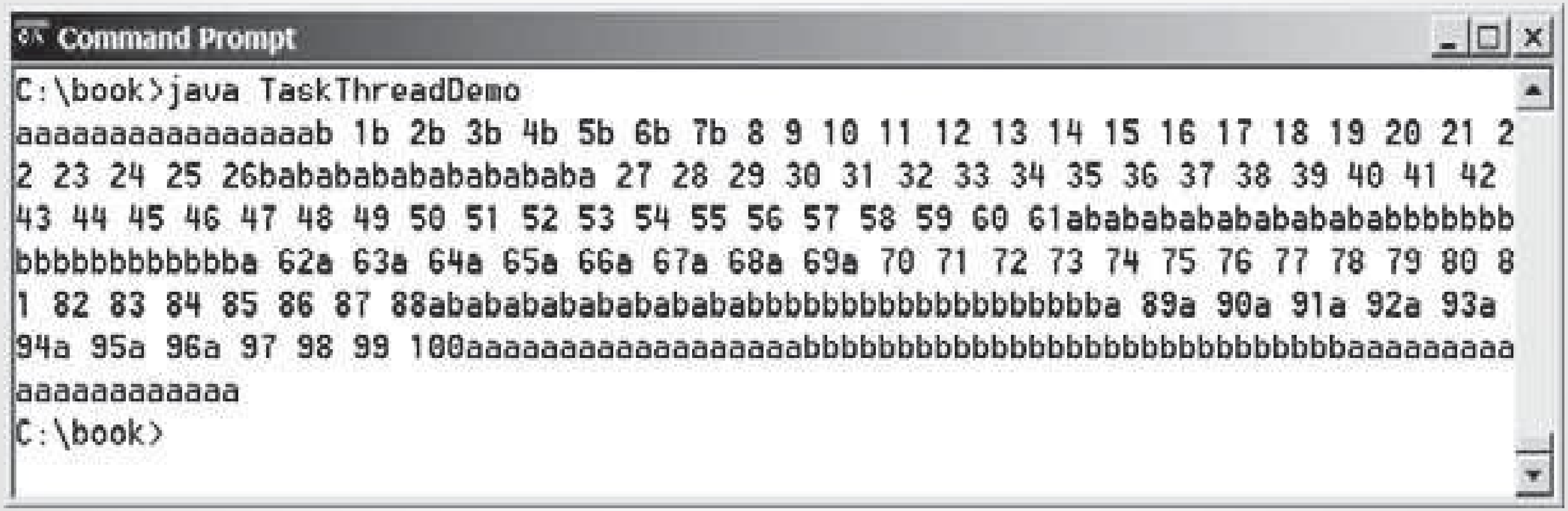
(b)

30.3 Creating Tasks and Threads

- A task must be executed in a thread.
- The Thread class contains the constructors for creating threads and many useful methods for controlling threads.
- The JVM will execute the task by invoking the task’s run() method.

30.3 Creating Tasks and Threads

- When you run this program, the three threads will share the CPU and take turns printing letters and numbers on the console.



Command Prompt

```
C:\book>java TaskThreadDemo
aaaaaaaaaaaaaaaab 1b 2b 3b 4b 5b 6b 7b 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
2 23 24 25 26bababababababababa 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61abababababababababbbbbbb
bbbbbbbbbbbba 62a 63a 64a 65a 66a 67a 68a 69a 70 71 72 73 74 75 76 77 78 79 80 8
1 82 83 84 85 86 87 88abababababababababbbbbbbbbbbbbbbbbba 89a 90a 91a 92a 93a
94a 95a 96a 97 98 99 100aaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaa
aaaaaaaaaaaa
C:\book>
```

30.3 Creating Tasks and Threads

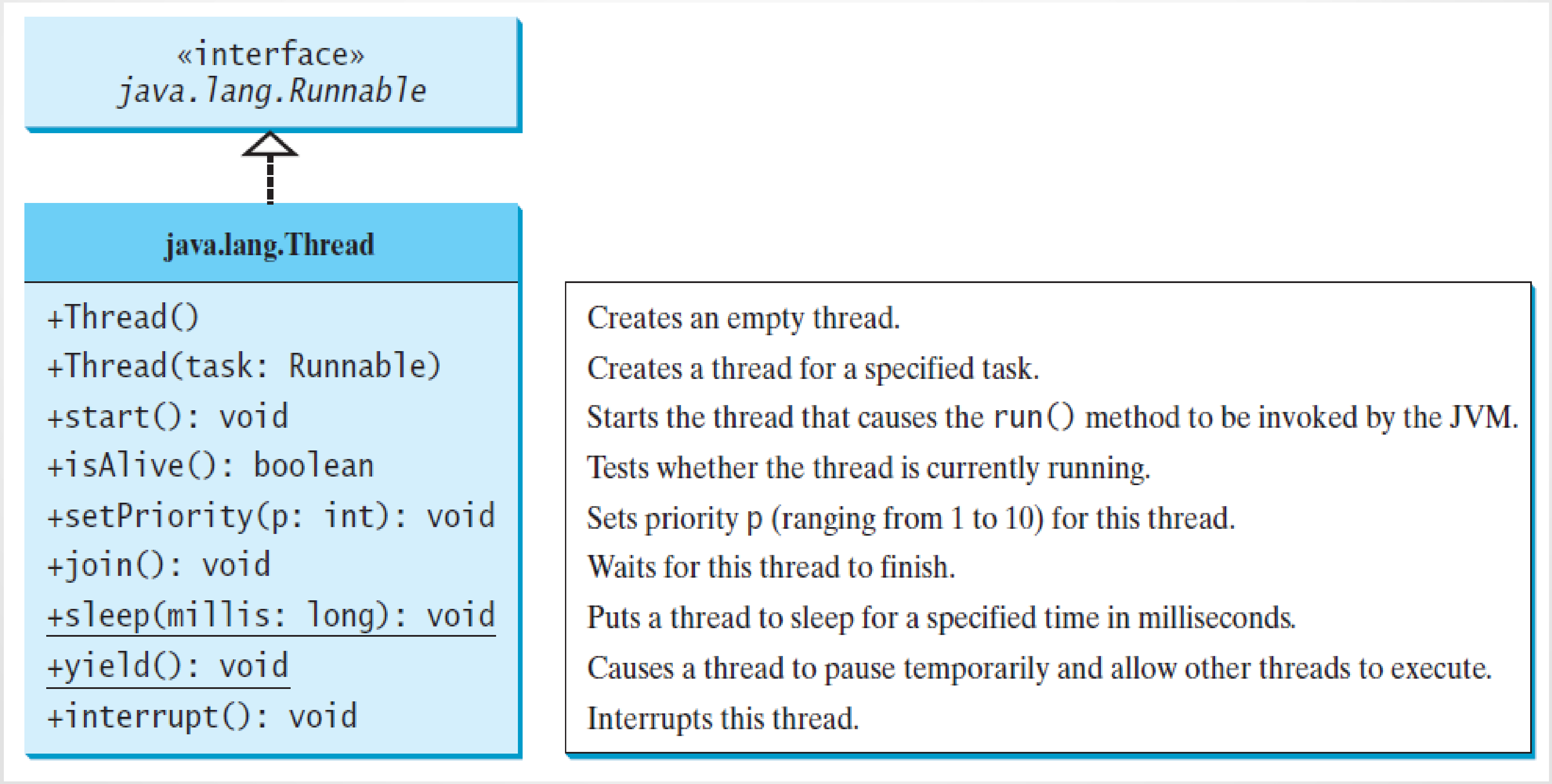
```java
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}

// The task for printing a character a specified number of times
class PrintChar implements Runnable {
  private char charToPrint; // The character to print
  private int times; // The number of times to repeat

  /** Construct a task with a specified character and number of
   *  times to print the character
   */
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }

  @Override /** Override the run() method to tell the system
   *  what task to perform
   */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
  private int lastNum;

  /** Construct a task for printing 1, 2, ..., n */
  public PrintNum(int n) {
    lastNum = n;
  }

  @Override /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
    }
  }
}
```

30.3 Creating Tasks and Threads

- When the run() method completes, the thread terminates.
- Because the first two tasks, printA and printB, have similar functionality, they can be defined in one task class PrintChar (lines 21–41).

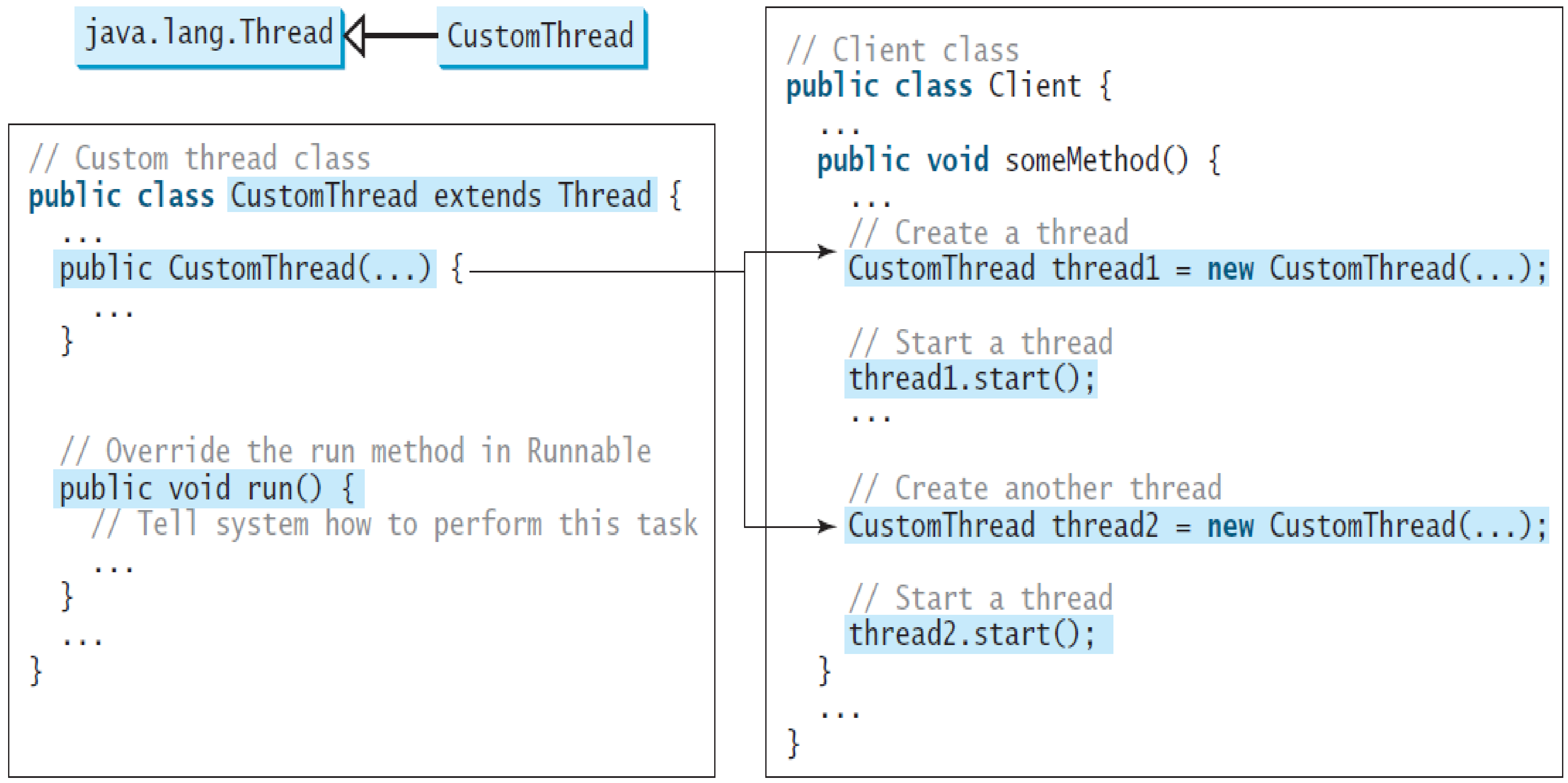
30.4 The Thread Class

- The Thread class contains the constructors for creating threads for tasks and the methods for controlling threads.



java.lang.Thread	
+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.

30.4 The Thread Class



```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }


  // Override the run method in Runnable
  public void run() {
    // Tell system how to perform this task
    ...
  }
  ...
}
```

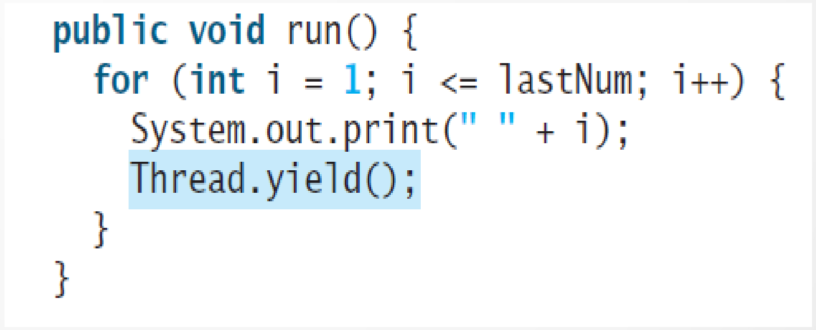
```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create a thread
    CustomThread thread1 = new CustomThread(...);

    // Start a thread
    thread1.start();
    ...

    // Create another thread
    CustomThread thread2 = new CustomThread(...);

    // Start a thread
    thread2.start();
  }
  ...
}
```

(a) (b)

FIGURE 30.5 Define a thread class by extending the Thread class.

30.4 The Thread Class

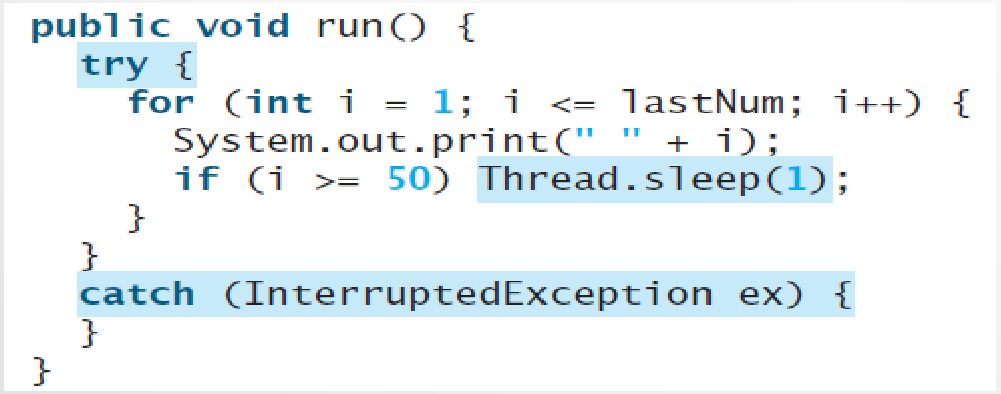
- You can use the yield() method to temporarily release time for other threads.
- For example, suppose you modify the code in the run() method in lines 53–57 for PrintNum in Listing 30.1 as follows:



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

30.4 The Thread Class

- Every time a number is printed, the thread of the print100 task is yielded to other threads.
- The sleep(long millis) method puts the thread to sleep for a specified time in milliseconds to allow other threads to execute.
- For example, suppose you modify the code in lines 53–57 in Listing 30.1, as follows:



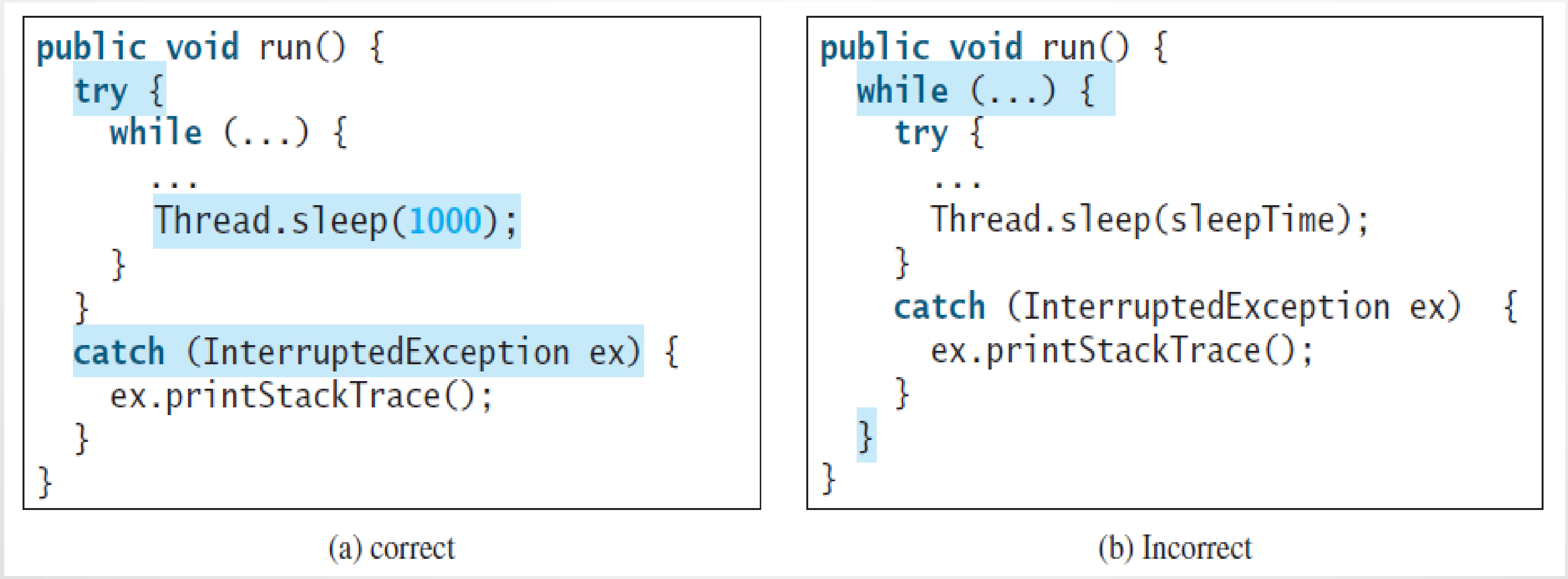
```
public void run() {
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i >= 50) Thread.sleep(1);
    }
  }
  catch (InterruptedException ex) {
  }
}
```

30.4 The Thread Class

- Every time a number (>= 50) is printed, the thread of the print100 task is put to sleep for 1 millisecond.
- The sleep method may throw an InterruptedException, which is a checked exception.
- Such an exception may occur when a sleeping thread’s interrupt() method is called.

30.4 The Thread Class

- If a sleep method is invoked in a loop, you should wrap the loop in a try-catch block, as shown in (a) below.
- If the loop is outside the try-catch block, as shown in (b), the thread may continue to execute even though it is being interrupted.



```
public void run() {
  try {
    while (...) {
      ...
      Thread.sleep(1000);
    }
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

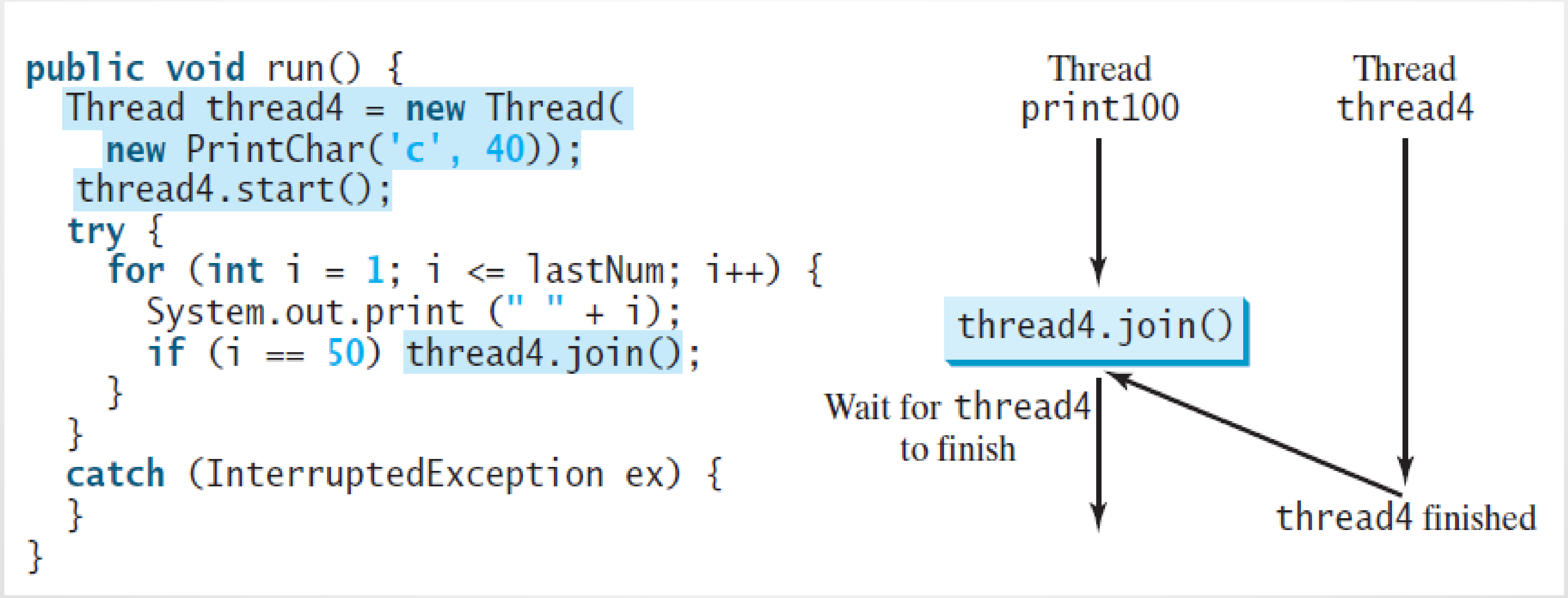
(a) correct

```
public void run() {
  while (...) {
    try {
      ...
      Thread.sleep(sleepTime);
    }
    catch (InterruptedException ex)  {
      ex.printStackTrace();
    }
  }
}
```

(b) Incorrect

30.4 The Thread Class

- 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 Listing 30.1 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) {
  }
}
```

30.4 The Thread Class

- The numbers from 50 to 100 are printed after thread thread4 is finished.
- Java assigns every thread a priority.
- You can increase or decrease the priority of any thread by using the setPriority method, and you can get the thread’s priority by using the getPriority method.
- Priorities are numbers ranging from 1 to 10.
- The Thread class has the int constants MIN_PRIORITY, NORM_PRIORITY, and MAX_PRIORITY, representing 1, 5, and 10, respectively.
- The priority of the main thread is Thread.NORM_PRIORITY.

30.4 The Thread Class

- The JVM always picks the currently runnable thread with the highest priority.
- A lowerpriority thread can run only when no higher-priority threads are running.
- If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue.
- This is called round-robin scheduling.
- For example, suppose you insert the following code in line 16 in Listing 30.1:
  - thread3.setPriority(Thread.MAX_PRIORITY);
- The thread for the print100 task will be finished first.

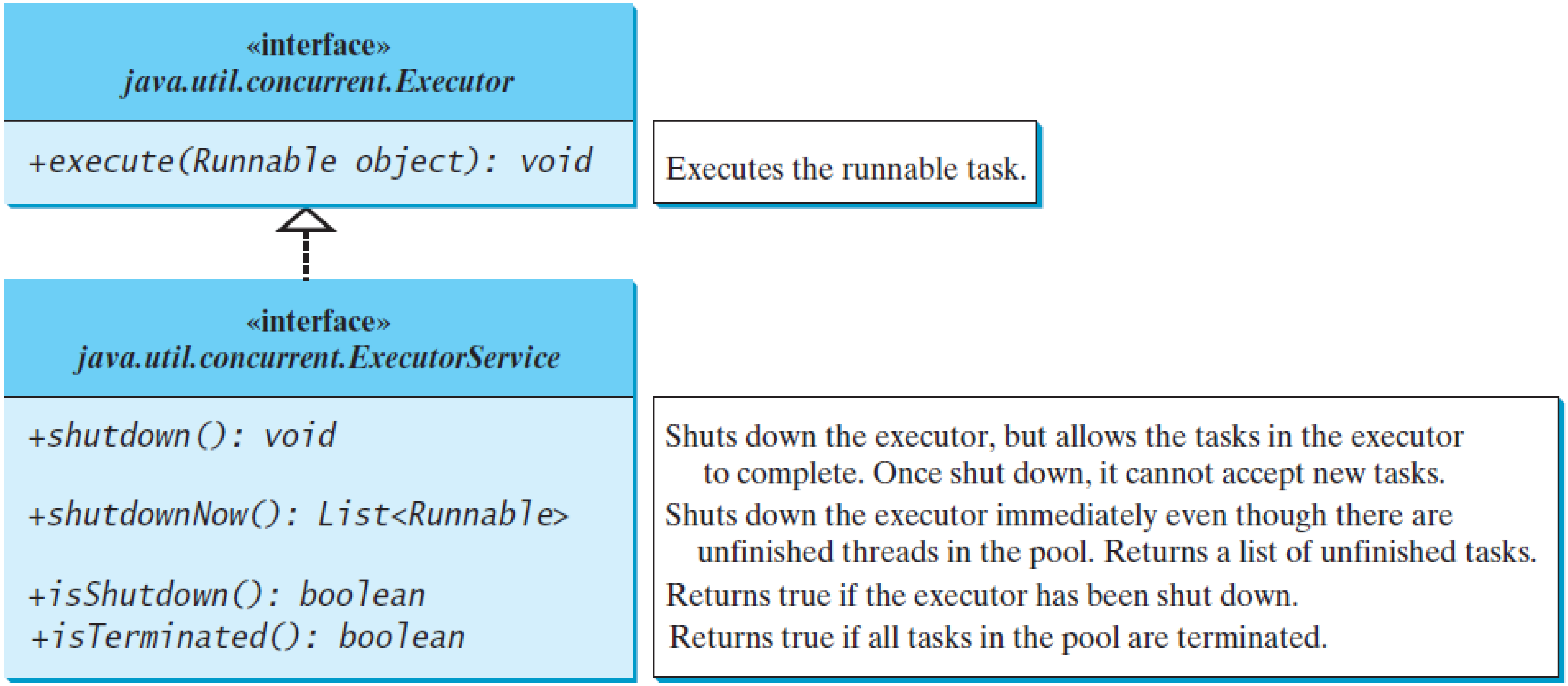
30.6 Thread Pools

- A thread pool can be used to execute tasks efficiently.
- In Section 30.3, Creating Tasks and Threads, you learned how to define a task class by implementing java.lang.Runnable, and how to create a thread to run a task like this:
  - Runnable task = new TaskClass(task);
  - new Thread(task).start();
- This approach is convenient for a single task execution, but it is not efficient for a large number of tasks because you have to create a thread for each task.

30.6 Thread Pools

- Starting a new thread for each task could limit throughput and cause poor performance.
- Using a thread pool is an ideal way to manage the number of tasks executing concurrently.
- Java provides the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks.

30.6 Thread Pools

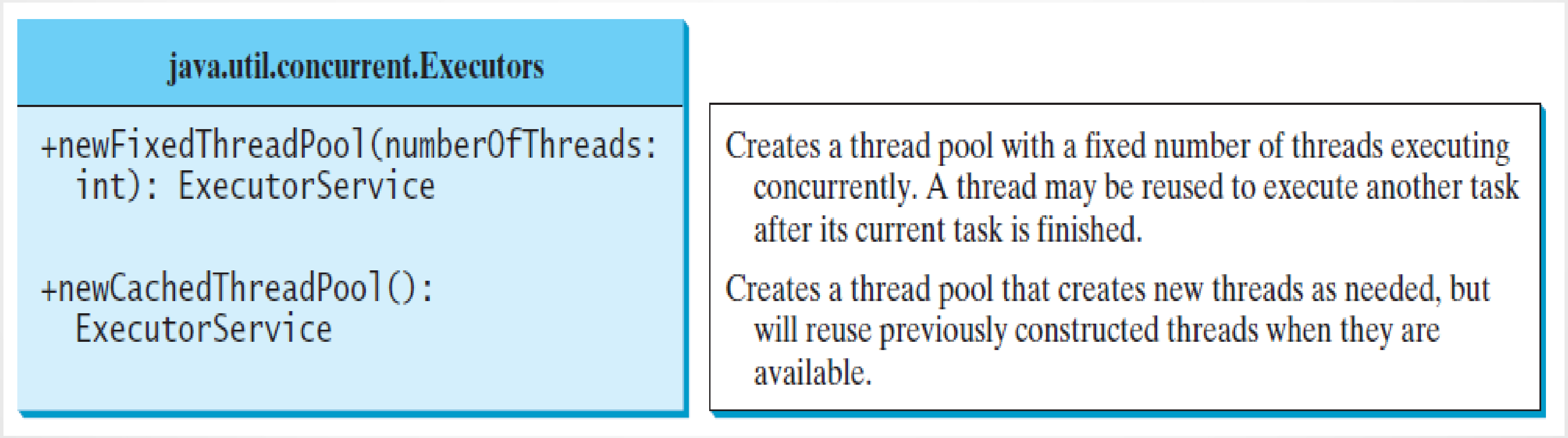

«interface»
java.util.concurrent.Executor
+execute(Runnable object): void
Executes the runnable task.
«interface»
java.util.concurrent.ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+isTerminated(): boolean
Shuts down the executor, but allows the tasks in the 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.

30.6 Thread Pools

- The newFixedThreadPool(int) method creates a fixed number of threads in a pool.
- If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it if all the threads in the pool are not idle and there are tasks waiting for execution.
- The newCachedThreadPool() method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution.

30.6 Thread Pools

- A thread in a cached pool will be terminated if it has not been used for 60 seconds.
- A cached pool is efficient for many short tasks.



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.

30.6 Thread Pools

```java
import java.util.concurrent.*;

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Shut down the executor
    executor.shutdown();
  }
}
```

30.6 Thread Pools

- The executor creates three threads to execute three tasks concurrently.
- Suppose that you replace line 6 with
  - ExecutorService executor = Executors.newFixedThreadPool(1);

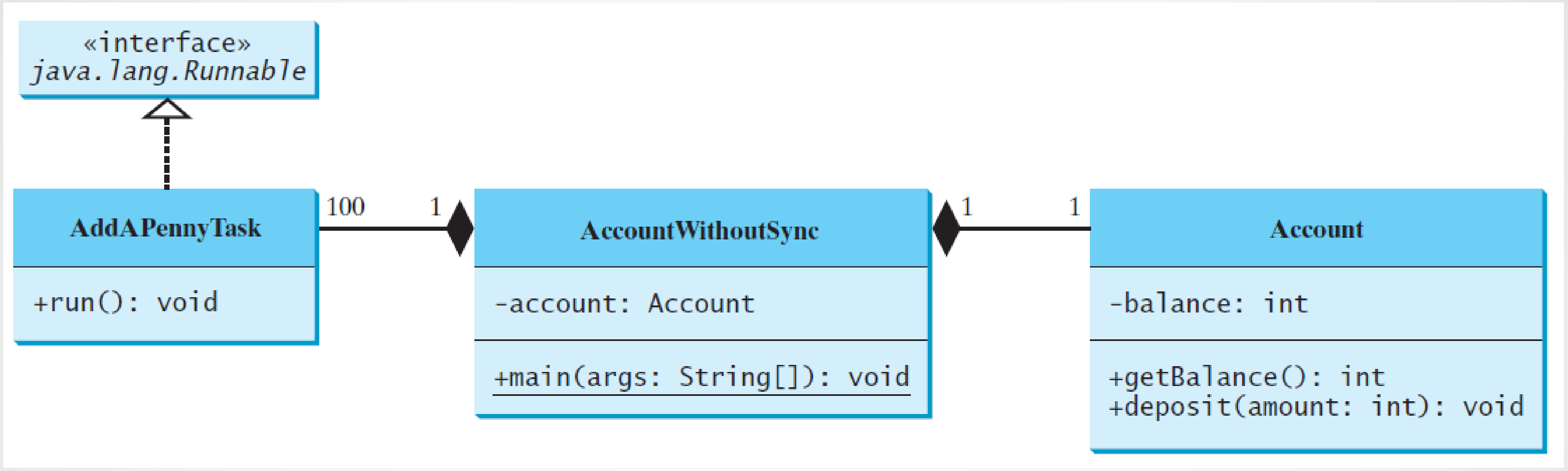
30.6 Thread Pools

- What will happen? The three runnable tasks will be executed sequentially because there is only one thread in the pool.
- Suppose you replace line 6 with
  - ExecutorService executor = Executors.newCachedThreadPool();
- What will happen? New threads will be created for each waiting task, so all the tasks will be executed concurrently.
- The shutdown() method in line 14 tells the executor to shut down.
- No new tasks can be accepted, but any existing tasks will continue to finish.

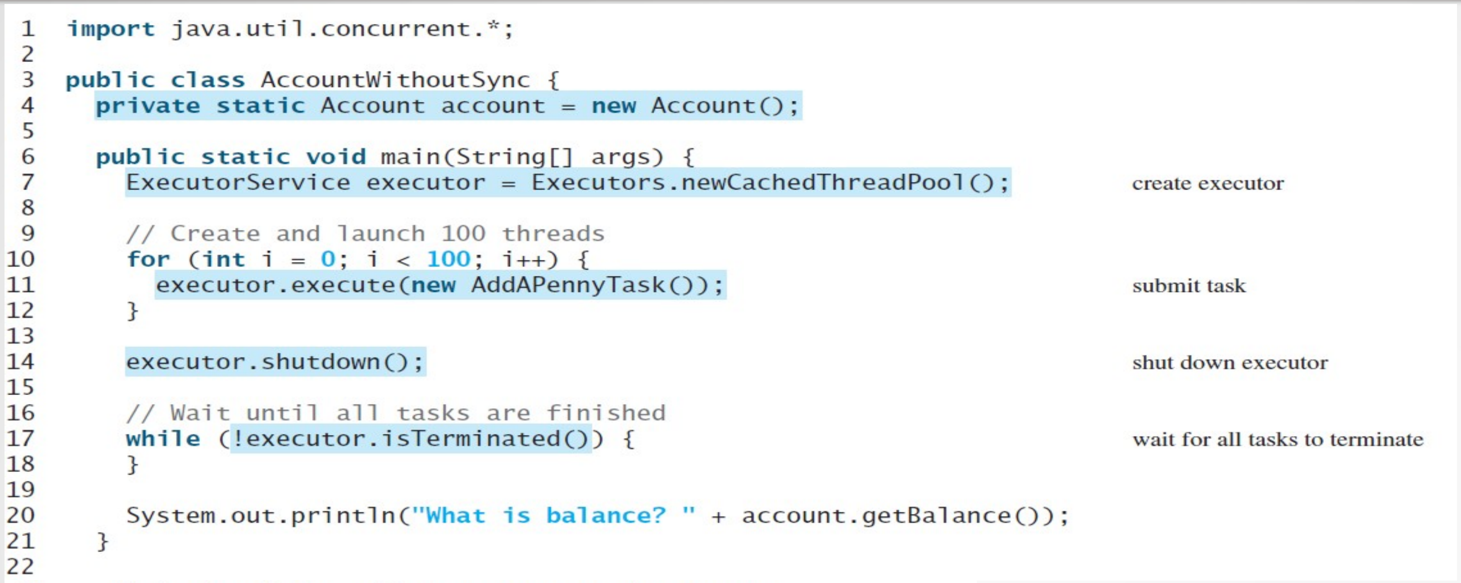
30.7 Thread Synchronization

- Thread synchronization is to coordinate the execution of the dependent threads.
- A shared resource may become corrupted if it is accessed simultaneously by multiple threads.
- Suppose you create and launch 100 threads, each of which adds a penny to an account.
- Define a class named Account to model the account, a class named AddAPennyTask to add a penny to the account, and a main class that creates and launches threads.

30.7 Thread Synchronization



30.7 Thread Synchronization



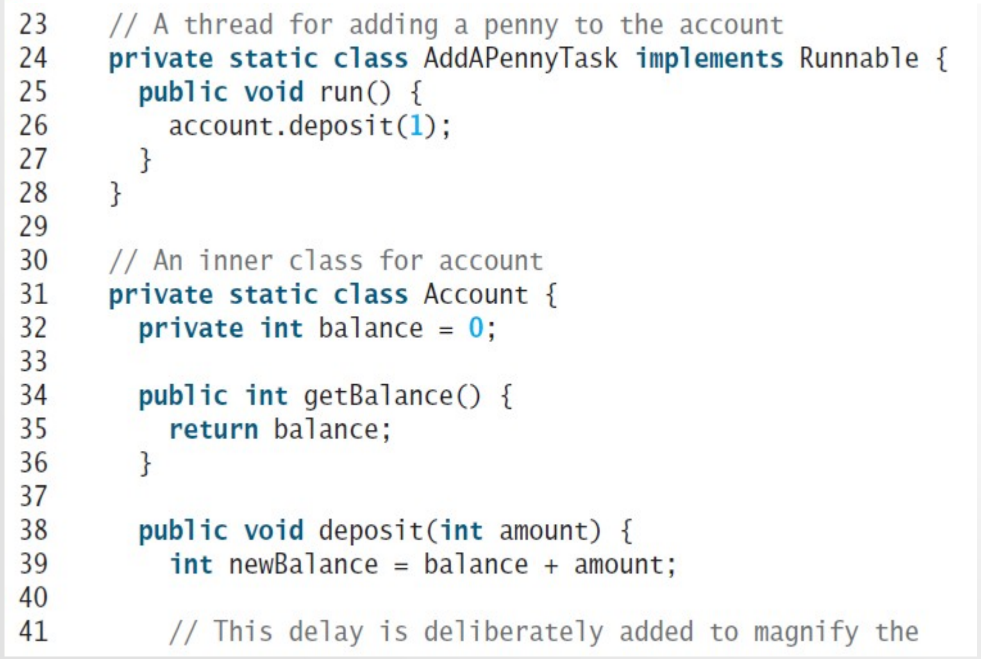
```java
import java.util.concurrent.*;

public class AccountWithoutSync {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

```

create executor

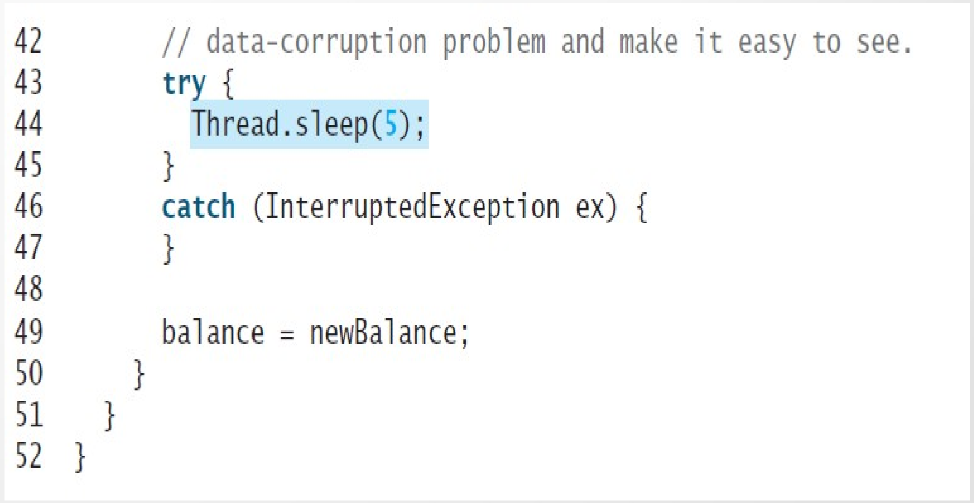
submit task

shut down executor

wait for all tasks to terminate



```java
  // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

  // An inner class for account
  private static class Account {
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      int newBalance = balance + amount;

      // This delay is deliberately added to magnify the
```



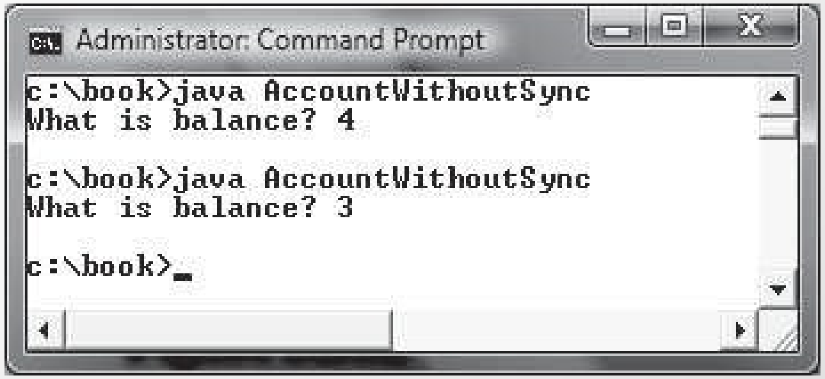
```java
      // data-corruption problem and make it easy to see.
      try {
        Thread.sleep(5);
      }
      catch (InterruptedException ex) {
      }

      balance = newBalance;
    }
  }
}
```

30.7 Thread Synchronization

- The isTerminated() method (line 17) is used to test whether the thread is terminated.
- The balance of the account is initially 0 (line 32).
- When all the threads are finished, the balance should be 100 but the output is unpredictable.
- This demonstrates the data-corruption problem that occurs when all the threads have access to the same data source simultaneously.

30.7 Thread Synchronization

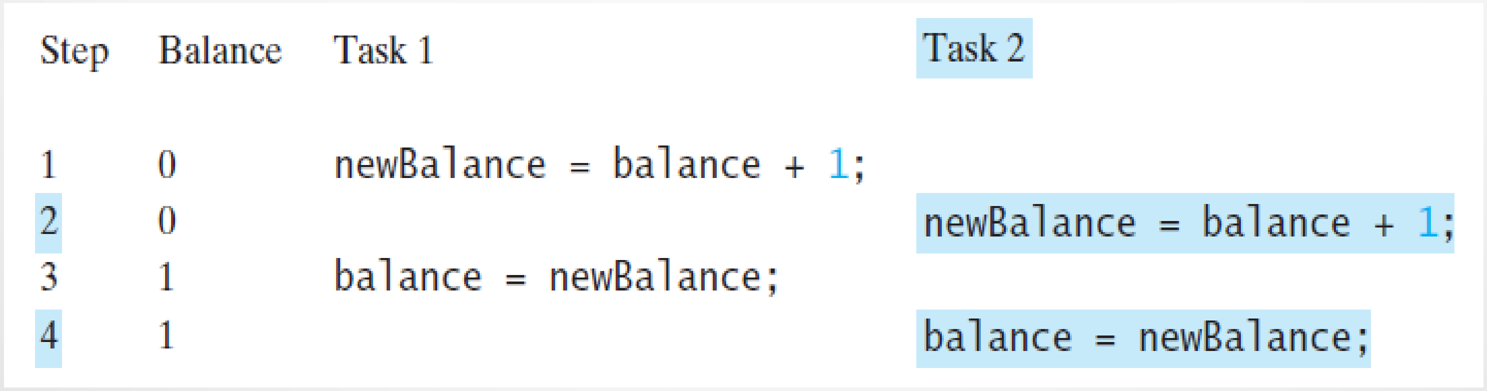
- Lines 39–49 could be replaced by one statement:
  - balance = balance + amount;
- It is highly unlikely, although plausible, that the problem can be replicated using this single statement.
- The statements in lines 39–49 are deliberately designed to magnify the datacorruption problem and make it easy to see.



```
c:\book>java AccountWithoutSync
What is balance? 4

c:\book>java AccountWithoutSync
What is balance? 3

c:\book>_
```

30.7 Thread Synchronization

- If you run the program several times but still do not see the problem, increase the sleep time in line 44.
- This will increase the chances for showing the problem of data inconsistency.
- What, then, caused the error in this program? A possible scenario is shown in Figure 30.11.



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

30.7 Thread Synchronization

- The effect of this scenario is that Task 1 does 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 a 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.

30.7.1 The synchronized Keyword

- To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the **critical region**.
- The critical region in Listing 30.4 is the entire deposit method.
- You can use the keyword **synchronized** 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 30.4.

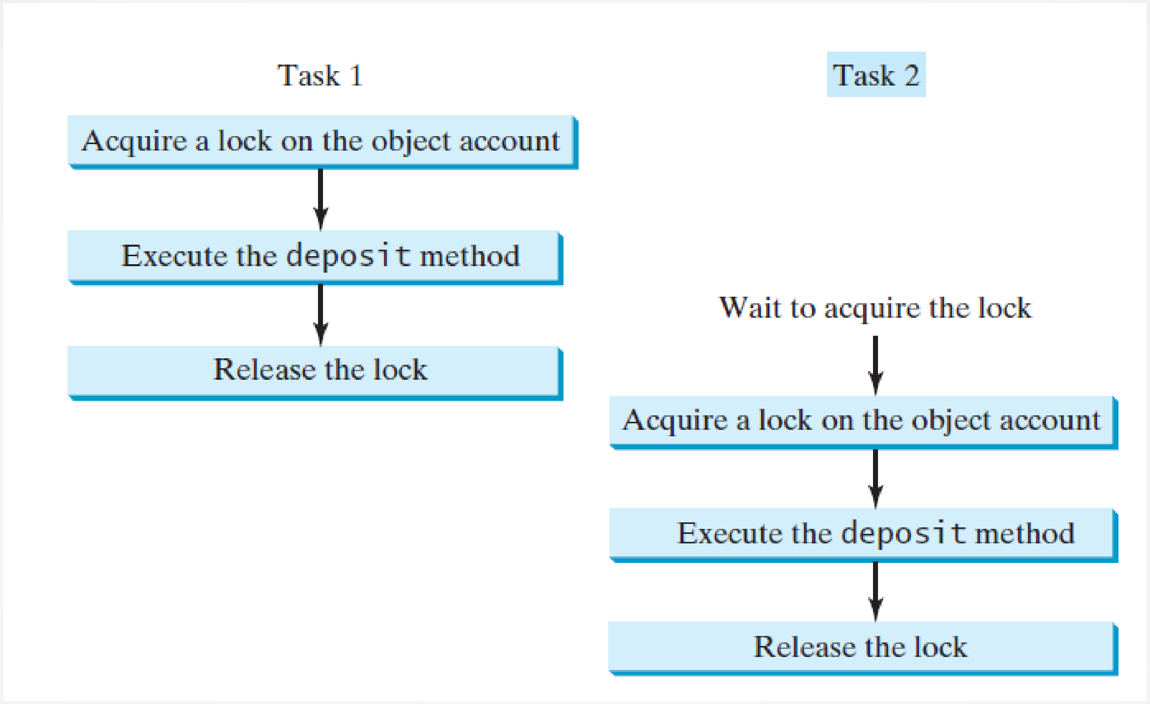
30.7.1 The synchronized Keyword

- One approach is to make Account thread-safe by adding the keyword synchronized in the deposit method in line 38, as follows:
  - **public synchronized void deposit(double amount)**
- A synchronized method acquires a lock before it executes.
- A lock is a mechanism for exclusive use of a resource.
- 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.

30.7.1 The synchronized Keyword

- 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.
- With the deposit method synchronized, the preceding scenario cannot happen.
- If Task 1 enters the method, Task 2 is blocked until Task 1 finishes the method, as shown in Figure 30.12.

30.7.1 The synchronized Keyword

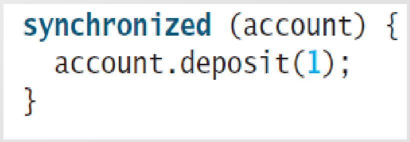

Task 1
Acquire a lock on the object account
Execute the deposit method
Release the lock
Task 2
Wait to acquire the lock
Acquire a lock on the object account
Execute the deposit method
Release the lock

30.7.2 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;
  - }

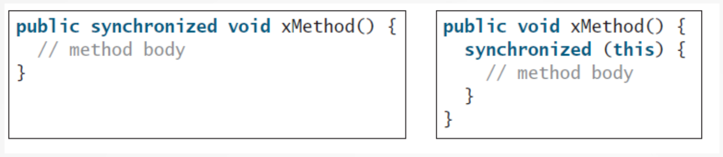
30.7.2 Synchronizing Statements

- 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.
- Synchronized statements enable you to synchronize part of the code in a method instead of the entire method.
- This increases concurrency.
- You can make Listing 30.4 thread-safe by placing the statement in line 26 inside a synchronized block:



```
synchronized (account) {
  account.deposit(1);
}
```

30.7.2 Synchronizing Statements



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

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

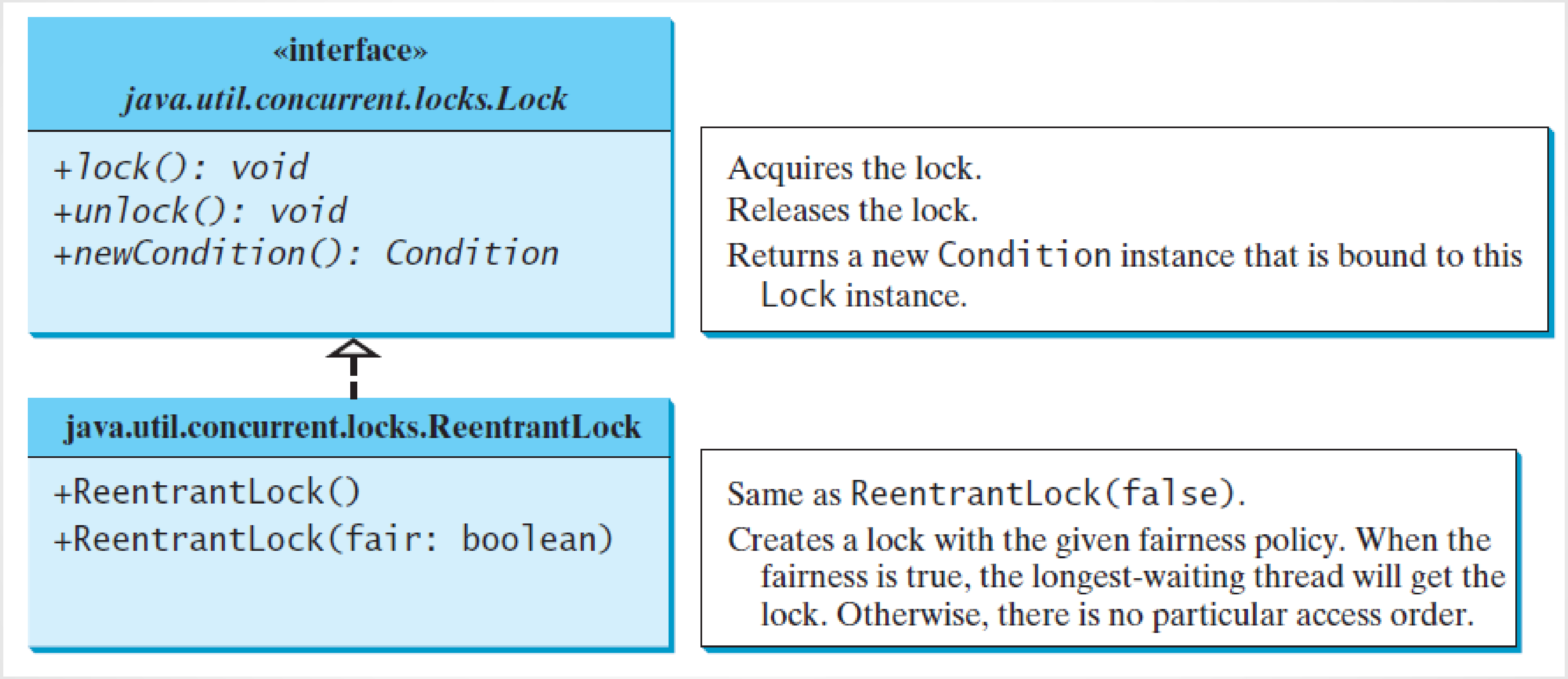
30.8 Synchronization Using Locks

- Locks and conditions can be explicitly used to synchronize threads.
- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.

30.8 Synchronization Using Locks

- Java enables you to acquire locks explicitly, which give you more control for coordinating threads.
- A lock is an instance of the Lock interface, which defines the methods for acquiring and releasing locks, as shown in Figure 30.13.
- A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.

30.8 Synchronization Using Locks


«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.
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.

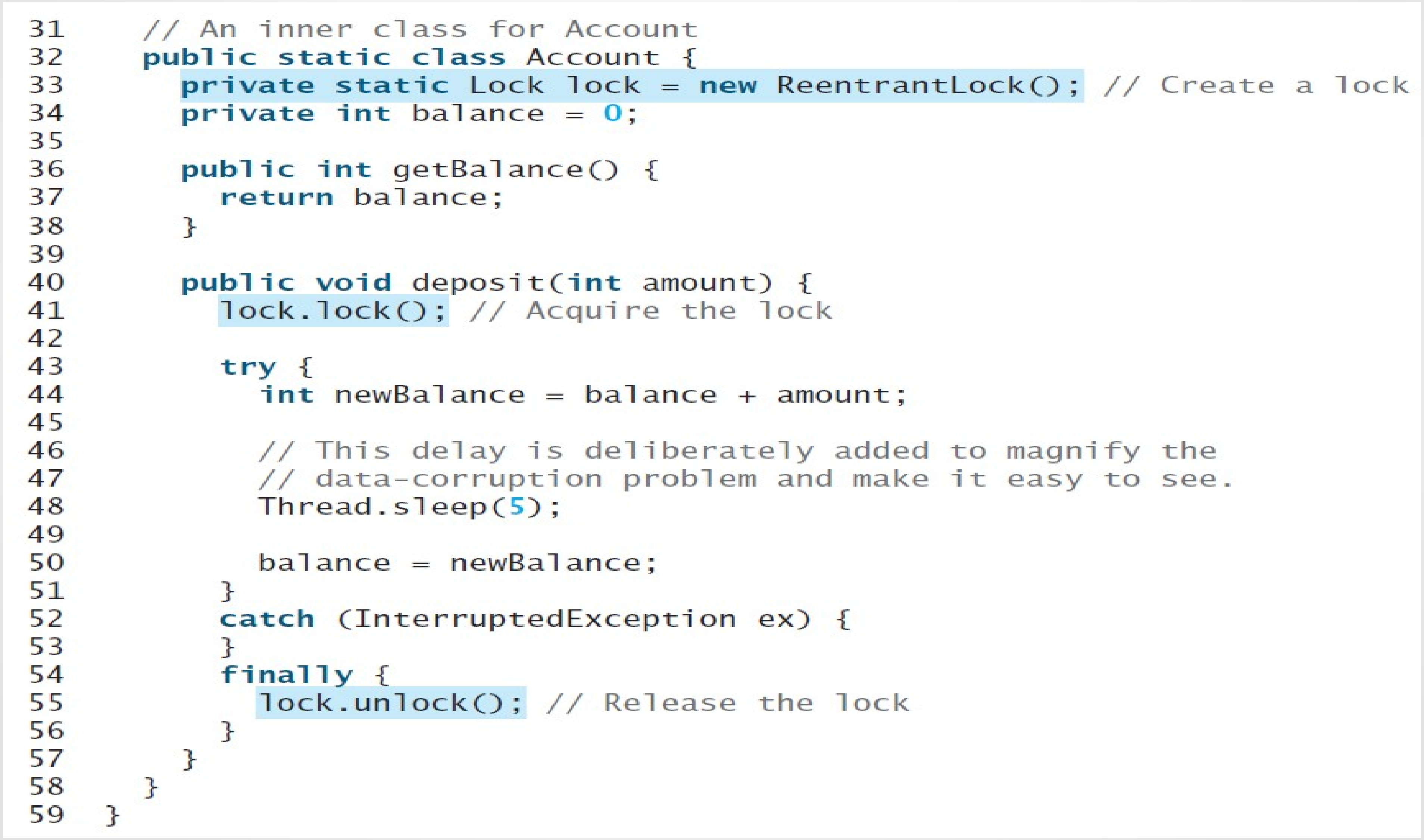
30.8 Synchronization Using Locks

- ReentrantLock is a concrete implementation of Lock for creating mutually exclusive locks.

30.8 Synchronization Using Locks

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

```

30.8 Synchronization Using Locks



```
  // An inner class for Account
  public static class Account {
    private static Lock lock = new ReentrantLock(); // Create a lock
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      lock.lock(); // Acquire the lock

      try {
        int newBalance = balance + amount;

        // This delay is deliberately added to magnify the
        // data-corruption problem and make it easy to see.
        Thread.sleep(5);

        balance = newBalance;
      }
      catch (InterruptedException ex) {
      }
      finally {
        lock.unlock(); // Release the lock
      }
    }
  }
}
```

30.8 Synchronization Using Locks

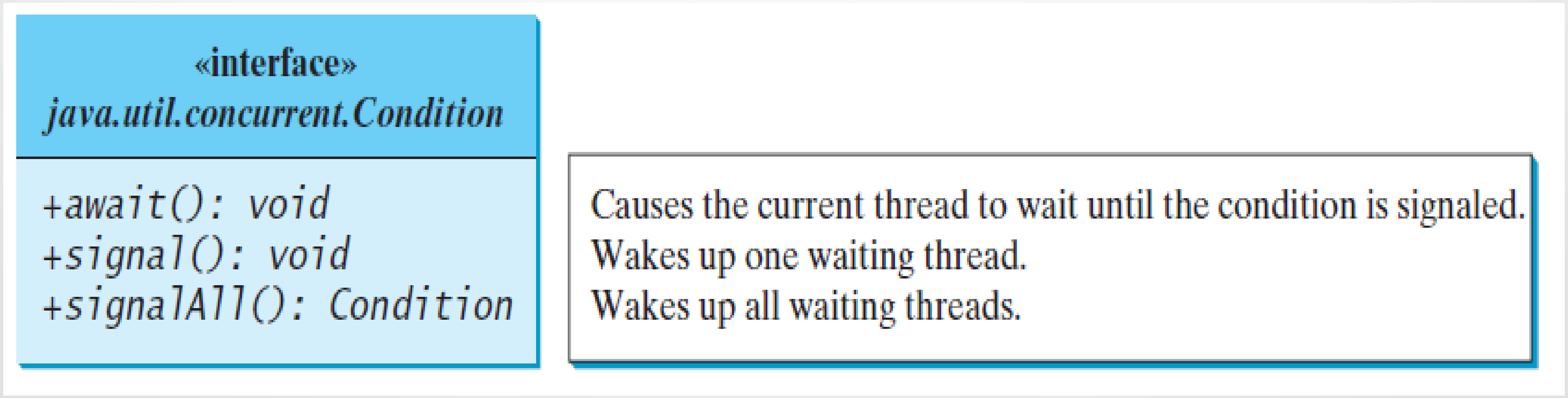
- Listing 30.5 can be implemented using a synchronize method for deposit rather than using a lock.
- In general, using synchronized methods or statements is simpler than using explicit locks for mutual exclusion.
- However, using explicit locks is more intuitive and flexible to synchronize threads with conditions, as you will see in the next section.

30.9 Cooperation among Threads

- Conditions on locks can be used to coordinate thread interactions.
- Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion of multiple threads in the critical region, but sometimes you also need a way for threads to cooperate.
- 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.

30.9 Cooperation among Threads

- 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
+signal(): void
+signalAll(): Condition
Causes the current thread to wait until the condition is signaled.
Wakes up one waiting thread.
Wakes up all waiting threads.

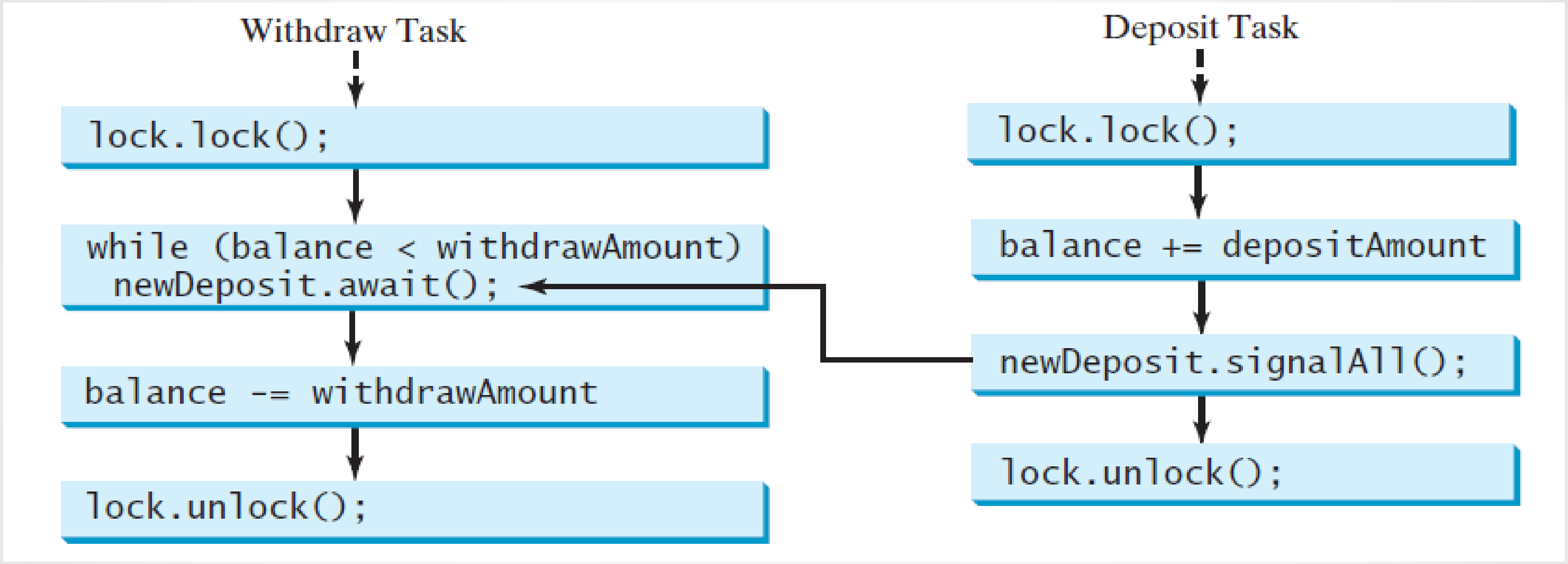
30.9 Cooperation among Threads

- Suppose that you create and launch two tasks: one that deposits into an account and one that withdraws from the same account.
- The withdraw task has to wait if the amount to be withdrawn is more than the current balance.
- Whenever new funds are deposited into the account, the deposit task notifies the withdraw thread to resume.
- If the amount is still not enough for a withdrawal, the withdraw thread has to continue to wait for a new deposit.

30.9 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.

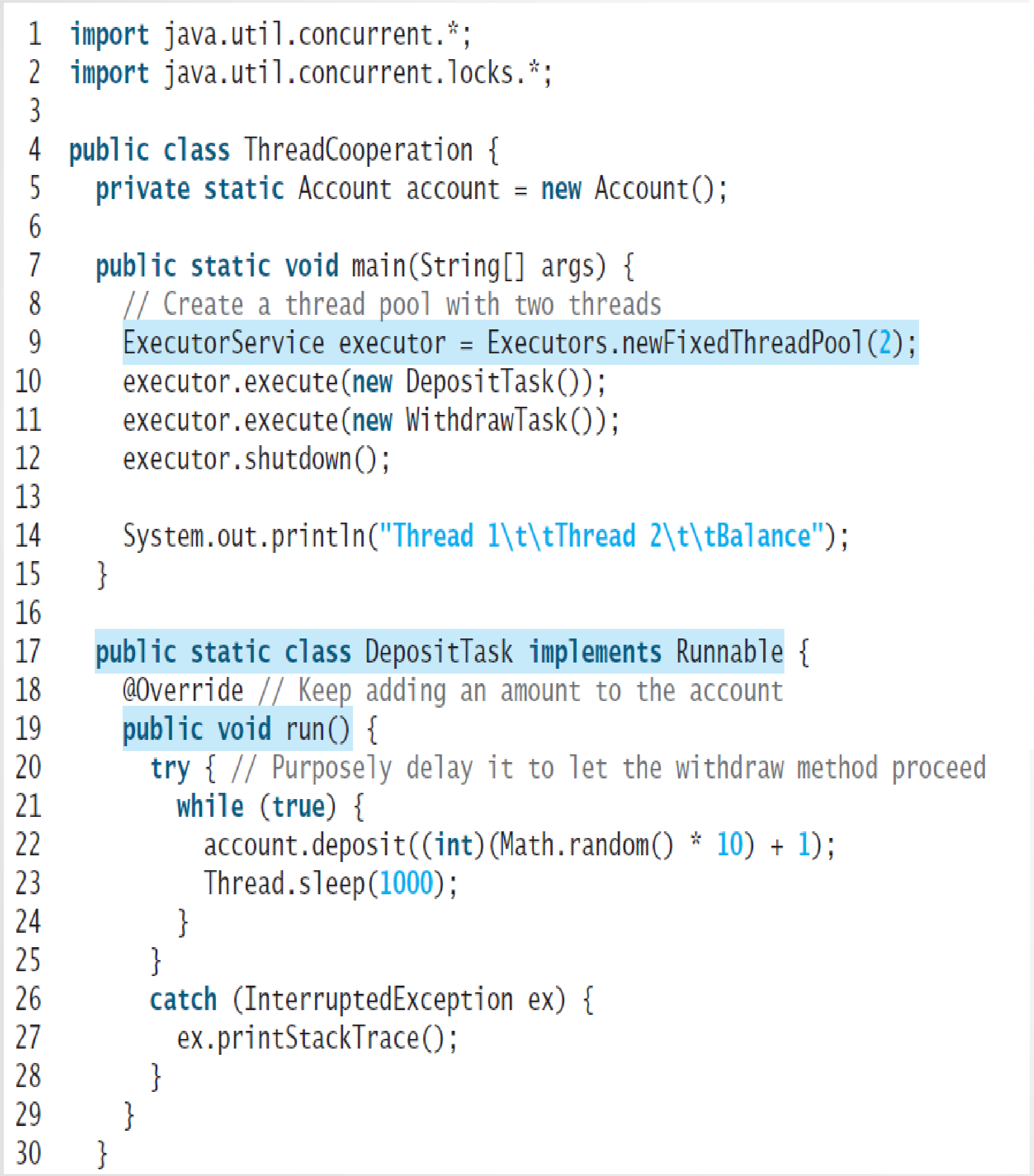
30.9 Cooperation among Threads



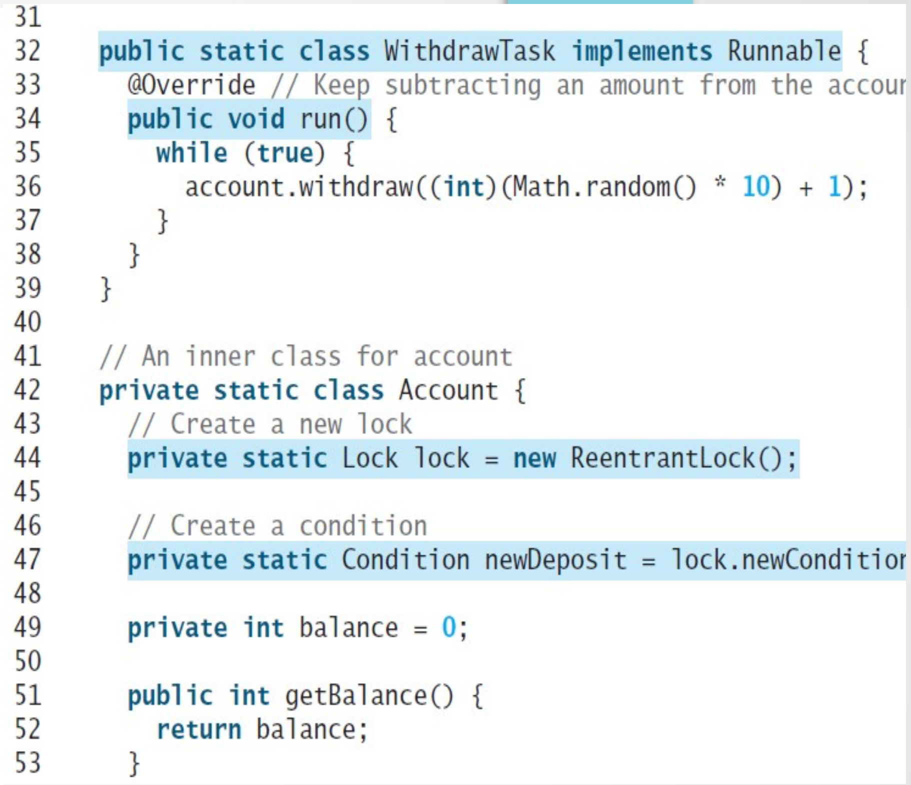
30.9 Cooperation among Threads

- To use a condition, you have to first obtain a lock.
- The await() method causes the thread to wait and automatically releases the lock on the condition.
- Once the condition is right, the thread reacquires the lock and continues executing.
- Assume that the initial balance is 0 and the amount to deposit and withdraw are randomly generated.

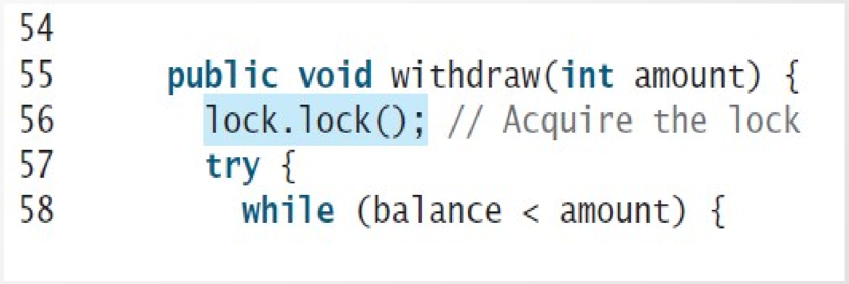
30.9 Cooperation among Threads



```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ThreadCooperation {
  private static Account account = new Account();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WithdrawTask());
    executor.shutdown();

    System.out.println("Thread 1\t\tThread 2\t\tBalance");
  }

  public static class DepositTask implements Runnable {
    @Override // Keep adding an amount to the account
    public void run() {
      try { // Purposely delay it to let the withdraw method proceed
        while (true) {
          account.deposit((int)(Math.random() * 10) + 1);
          Thread.sleep(1000);
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
```



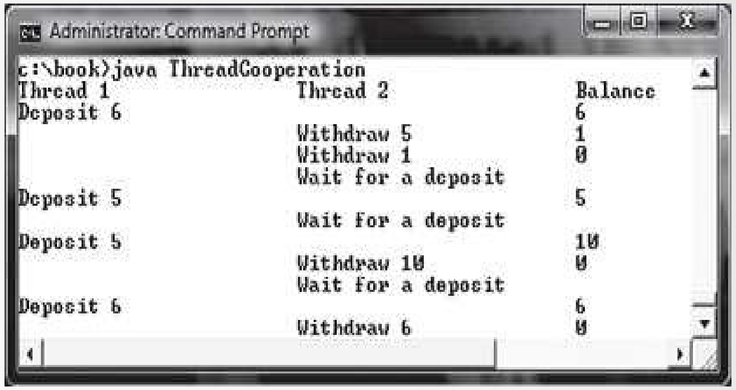
```java

  public static class WithdrawTask implements Runnable {
    @Override // Keep subtracting an amount from the accour
    public void run() {
      while (true) {
        account.withdraw((int)(Math.random() * 10) + 1);
      }
    }
  }

  // An inner class for account
  private static class Account {
    // Create a new lock
    private static Lock lock = new ReentrantLock();

    // Create a condition
    private static Condition newDeposit = lock.newConditior

    private int balance = 0;

    public int getBalance() {
      return balance;
    }
```



```java

    public void withdraw(int amount) {
      lock.lock(); // Acquire the lock
      try {
        while (balance < amount) {
```

30.9 Cooperation among Threads

```java
          System.out.println("\t\t\tWait for a deposit");
          newDeposit.await();
        }

        balance -= amount;
        System.out.println("\t\t\tWithdraw " + amount +
          "\t\t" + getBalance());
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      finally {
        lock.unlock(); // Release the lock
      }
    }

    public void deposit(int amount) {
      lock.lock(); // Acquire the lock
      try {
        balance += amount;
        System.out.println("Deposit " + amount +
          "\t\t\t\t\t" + getBalance());

        // Signal thread waiting on the condition
        newDeposit.signalAll();
      }
      finally {
        lock.unlock(); // Release the lock
      }
    }
  }
}
```



```
c:\book>java ThreadCooperation
Thread 1                 Thread 2                 Balance
Deposit 6                                         6
                         Withdraw 5               1
                         Withdraw 1               0
                         Wait for a deposit
Deposit 5                                         5
                         Wait for a deposit
Deposit 5                                         10
                         Withdraw 10              0
                         Wait for a deposit
Deposit 6                                         6
                         Withdraw 6               0
```

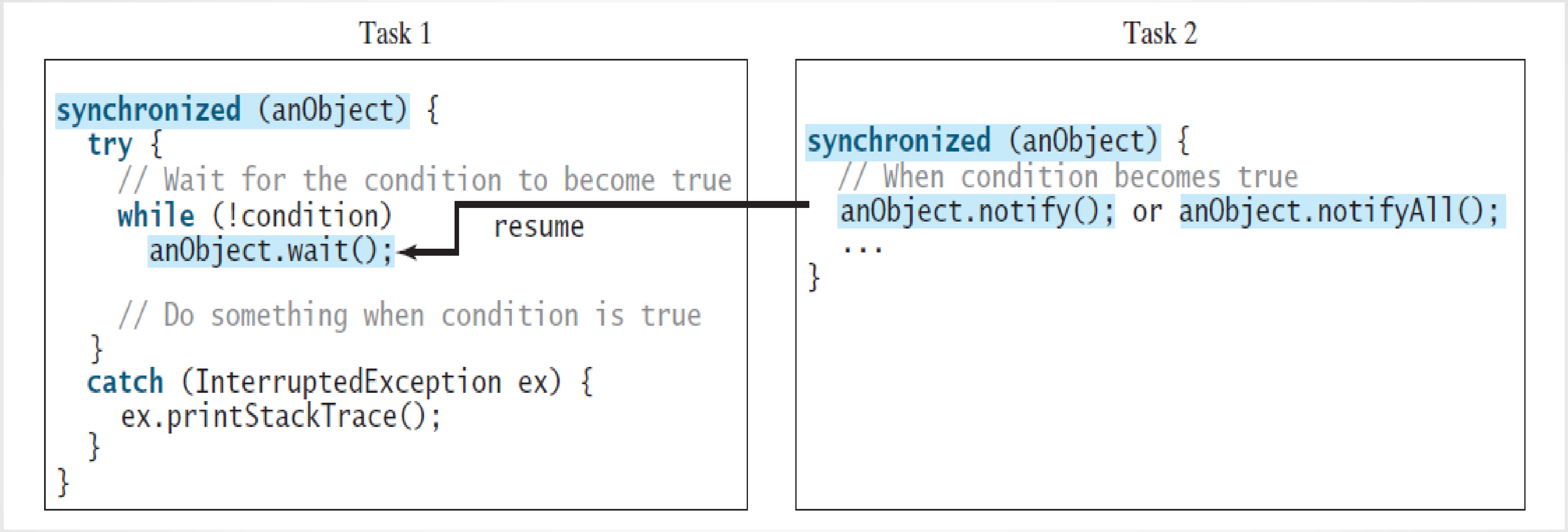
30.9 Cooperation among Threads

- 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 it 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.

30.9 Cooperation among Threads

- You can invoke the wait() method on the monitor object to release the lock so that some other thread can get in the monitor and perhaps change the monitor’s state.
- When the condition is right, the other thread can invoke the notify() or notifyAll() method to signal one or all waiting threads to regain the lock and resume execution.
- The template for invoking these methods is shown in Figure 30.17.

30.9 Cooperation among Threads



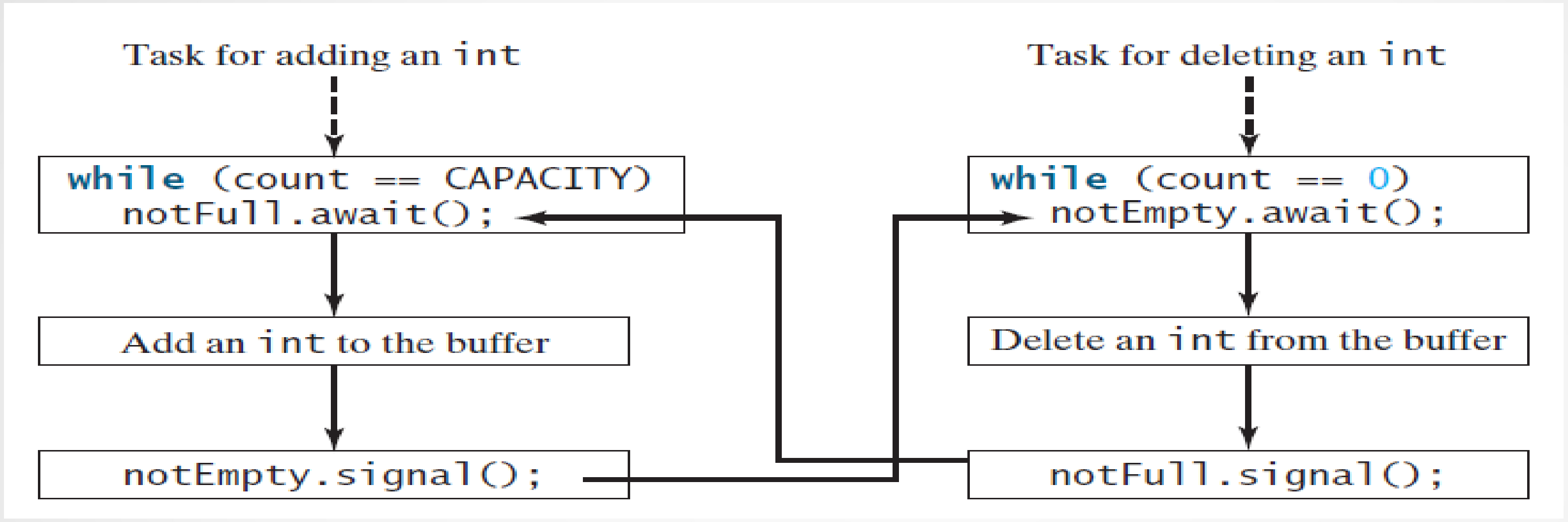
```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

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

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

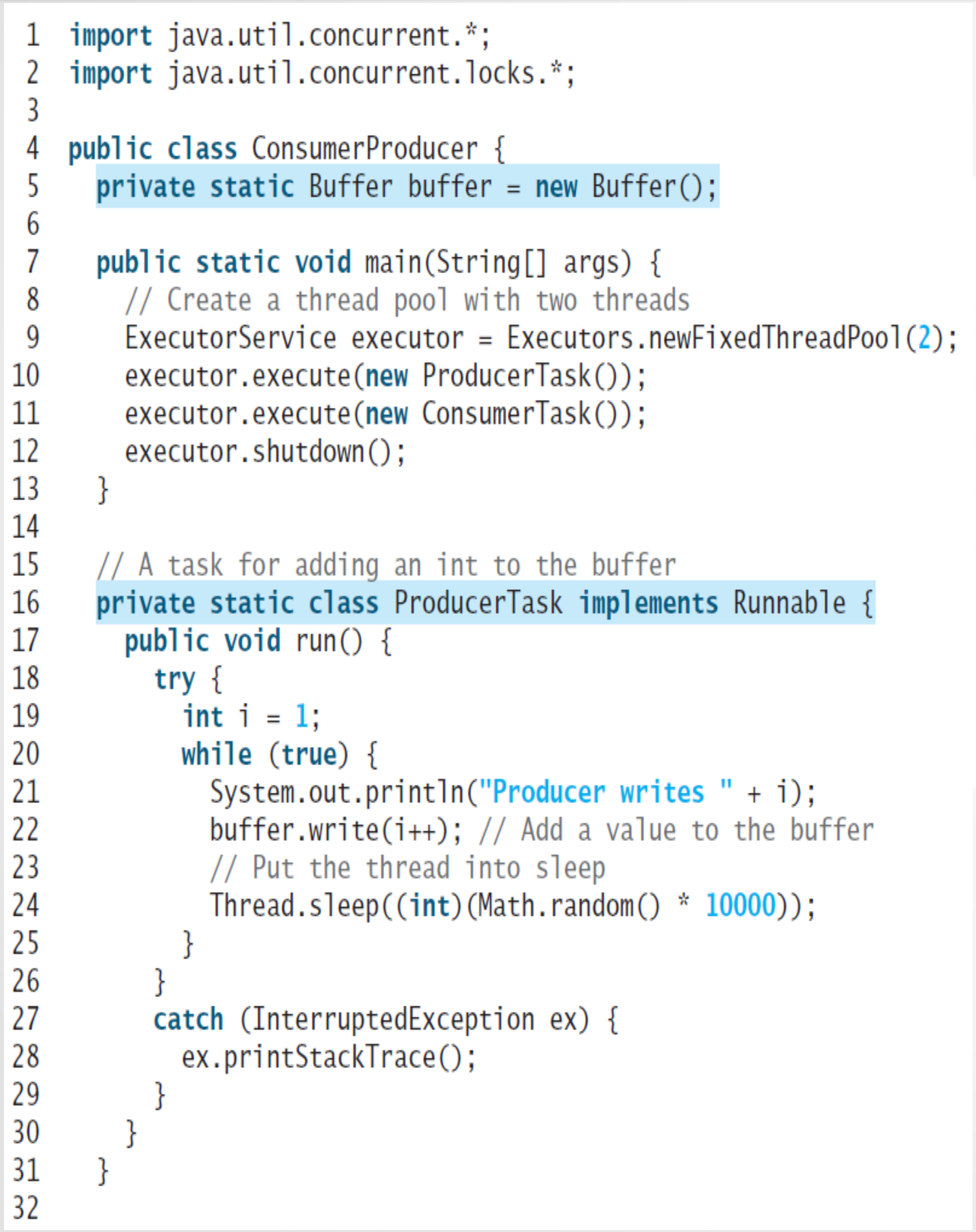
30.9 Cooperation among Threads

- 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.

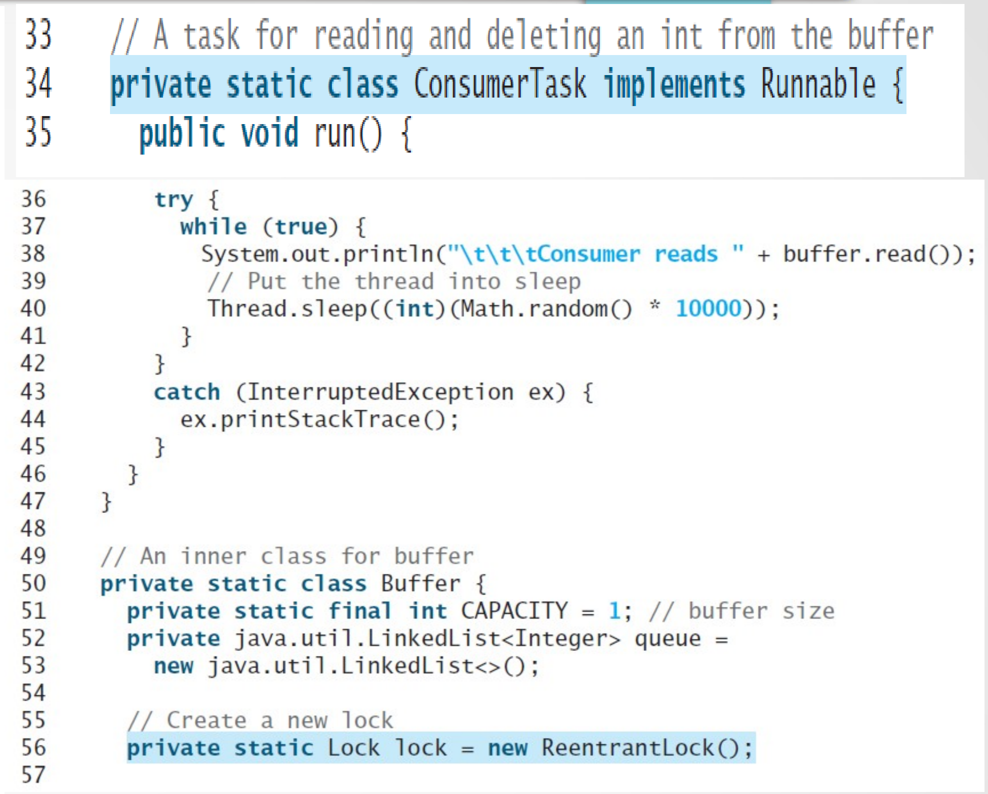
30.10 Case Study: Producer/Consumer



30.10 Case Study: Producer/Consumer



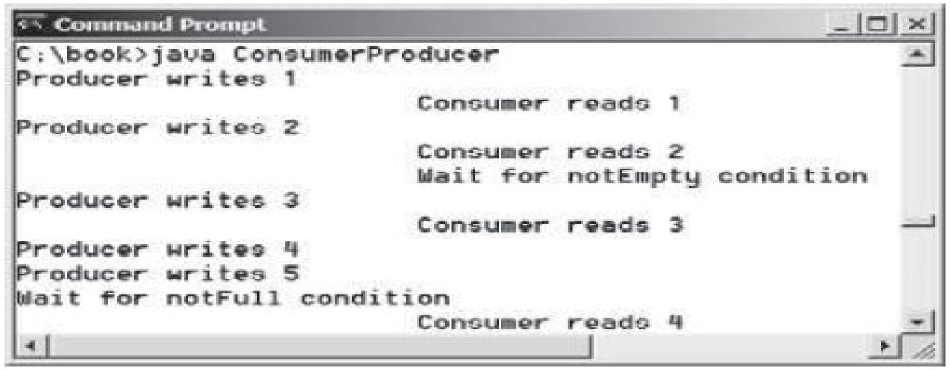
```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ConsumerProducer {
  private static Buffer buffer = new Buffer();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }

  // A task for adding an int to the buffer
  private static class ProducerTask implements Runnable {
    public void run() {
      try {
        int i = 1;
        while (true) {
          System.out.println("Producer writes " + i);
          buffer.write(i++); // Add a value to the buffer
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

```



```java
  // A task for reading and deleting an int from the buffer
  private static class ConsumerTask implements Runnable {
    public void run() {
      try {
        while (true) {
          System.out.println("\t\t\tConsumer reads " + buffer.read());
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  // An inner class for buffer
  private static class Buffer {
    private static final int CAPACITY = 1; // buffer size
    private java.util.LinkedList<Integer> queue =
      new java.util.LinkedList<>();

    // Create a new lock
    private static Lock lock = new ReentrantLock();

```

30.10 Case Study: Producer/Consumer

```java

    // Create two conditions
    private static Condition notEmpty = lock.newCondition();
    private static Condition notFull = lock.newCondition();

    public void write(int value) {
      lock.lock(); // Acquire the lock
      try {
        while (queue.size() == CAPACITY) {
          System.out.println("Wait for notFull condition");
          notFull.await();
        }

        queue.offer(value);
        notEmpty.signal(); // Signal notEmpty condition
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      finally {
        lock.unlock(); // Release the lock
      }
    }

    public int read() {
      int value = 0;
      lock.lock(); // Acquire the lock
      try {
        while (queue.isEmpty()) {
          System.out.println("\t\t\tWait for notEmpty condition");
          notEmpty.await();
        }

        value = queue.remove();
        notFull.signal(); // Signal notFull condition
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      finally {
        lock.unlock(); // Release the lock
        return value;
      }
    }
  }
}
```



```
C:\book>java ConsumerProducer
Producer writes 1
                        Consumer reads 1
Producer writes 2
                        Consumer reads 2
                        Wait for notEmpty condition
Producer writes 3
                        Consumer reads 3
Producer writes 4
Producer writes 5
Wait for notFull condition
                        Consumer reads 4
```