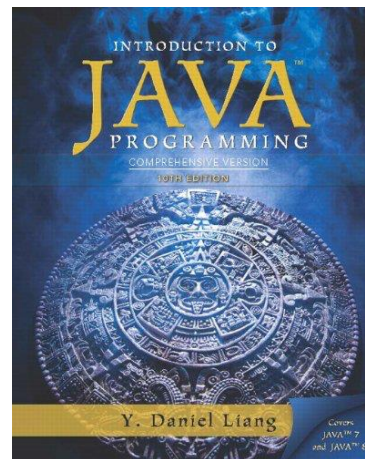


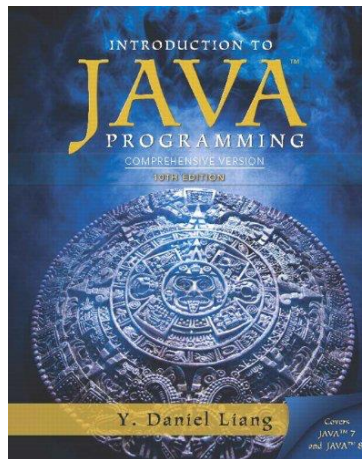
# Chapter 30

## Multithreading and Parallel Programming



## ➤ Supplemental Course Material

**These lecture notes are based on Chapter 30 of the following book:**



- **Introduction to Java Programming, Comprehensive Version (10<sup>th</sup> Edition)**

**Authors:** Y. Daniel Liang

**ISBN:** 978-0133761313

**Publisher:** Pearson, 10<sup>th</sup> Edition (2015)

- **N.B.: Chapter 30, “Multithreading and Parallel Programming”, will be available in PDF format on myCourses (Copyright permission obtained via McGill Libraries).**

## 30.1 Introduction / Motivation

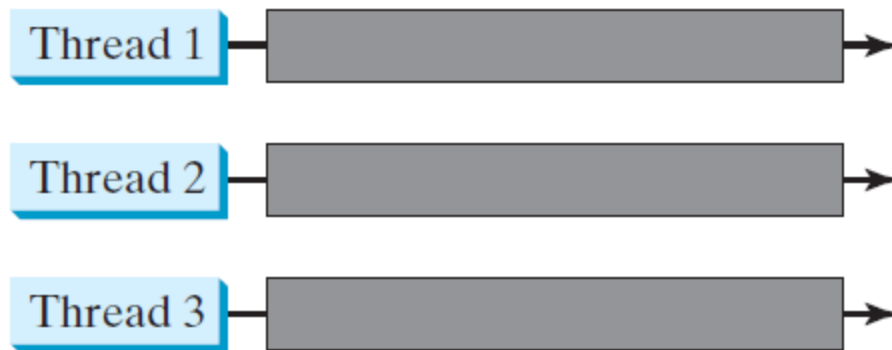
- One of the **powerful features** of **Java** is its **built-in support** for ***multithreading*** – the *concurrent running of multiple tasks within a program*.
- In **many programming languages**, you have to invoke **system-dependent procedures** and **functions** to implement multithreading.
- This chapter introduces the concept of **threads** and **how multithreading programs** can be **developed in Java**.
- An introduction to developing ***parallel programs*** in Java using the **Fork/Join Framework** is also given.

# Objectives

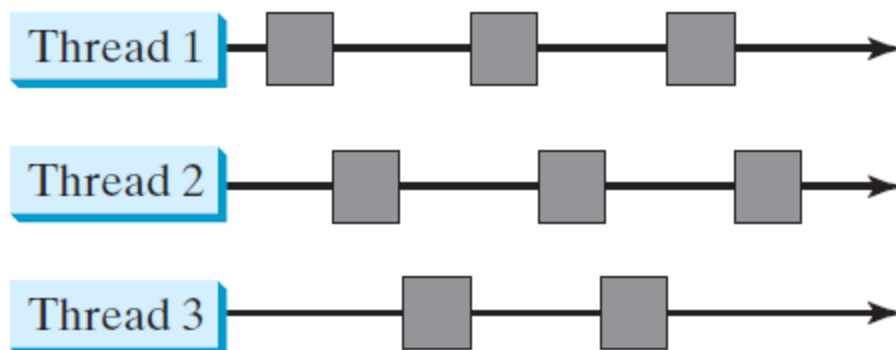
- To get an overview of multithreading (§30.2).
- To develop task classes by implementing the **Runnable** interface (§30.3).
- To create threads to run tasks using the **Thread** class (§30.3).
- To control threads using the methods in the **Thread** class (§30.4).
- To control animations using threads and use **Platform.runLater** to run the code in the 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).

## 30.2 Thread Concepts

Multiple  
threads on  
multiple  
CPUs



Multiple  
threads  
*sharing* a  
single CPU



## 30.3 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

## Important Note

- The `run()` method in a task specifies how to perform the task. This method is *automatically invoked* by the JVM. You should *not* invoke it. Invoking `run()` directly merely executes this method in the same thread; *no new thread is started*.

## What is the output of the following code?

// Test.java: Define threads using the Thread class

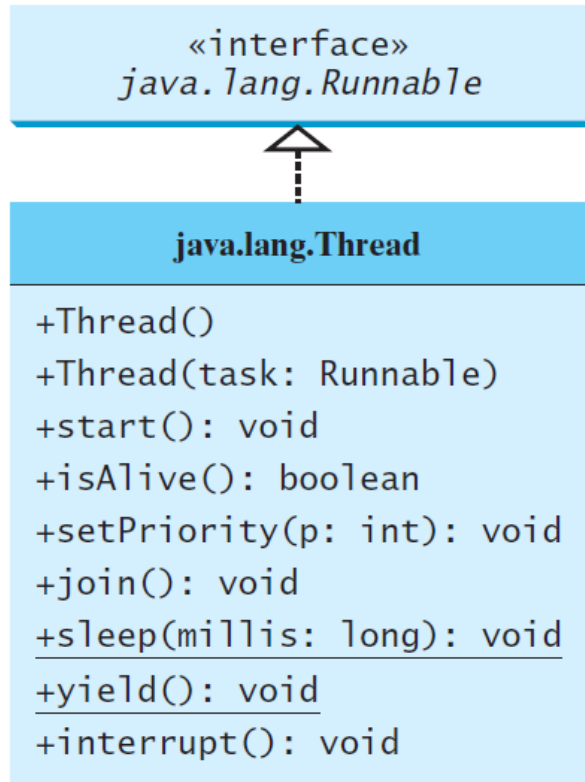
```
public class Test {
    /** Main method */
    public static void main(String[] args) {
        new Test();
    }
    public Test() {
        // Create threads
        PrintChar printA = new PrintChar('a', 4);
        PrintChar printB = new PrintChar('b', 4);
        // Start threads
        printA.run();
        printB.run();
    }
    class PrintChar implements Runnable {
        private char charToPrint; // The character to print
        private int times; // The times to repeat

        /** Construct a thread with specified character and number of
         times to print the character */
        public PrintChar(char c, int t) {
            charToPrint = c;
            times = t;
        }
        /** Override the run() method to tell the system
         what the thread will do */
        public void run() {
            for (int i = 0; i < times; i++)
                System.out.print(charToPrint);
        }
    }
}
```

- A. aaaabbbb
- B. bbbbaaaa
- C. character a and b are randomly printed
- D. abababab



# 30.4 The Thread Class



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.

## ⚡Note

- Since the **Thread** class implements **Runnable**, you could **define a class** that **extends Thread** and implements the **run** method, as shown in (a) below, and then create an object from the class and invoke its **start** method in a client program to start the thread, as shown in (b) below.
- However, **this approach is *not* recommended** because it mixes the task and mechanism of running the task.
- **Separating the task from the thread**, as shown earlier is a **preferred design**.

```

? java.lang.Thread ← CustomThread
?
? // 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
?         ...
?     }
? }

```

(a)

```

? // 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();
?     }
?     ...
? }

```

(b)

## 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.

# 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:

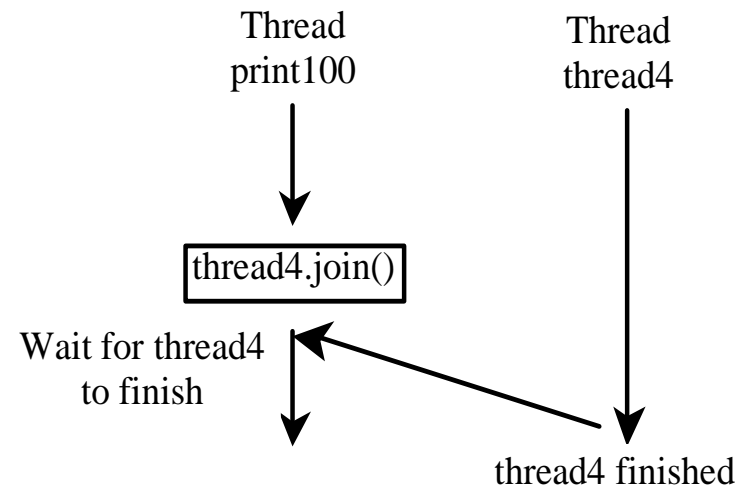
```
public void run() {  
    for (int i = 1; i <= lastNum; i++) {  
        System.out.print(" " + i);  
        try {  
            if (i >= 50) Thread.sleep(1);  
        }  
        catch (InterruptedException ex) {  
        }  
    }  
}
```

- Every time a number ( $\geq 50$ ) 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 `thread4` is **finished**.

# 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`

Which of the following expressions must be true if you create a thread using `Thread = new Thread(object)`?

- A. `object instanceof Thread`
- B. `object instanceof Frame`
- C. `object instanceof Applet`
- D. `object instanceof Runnable`



Which of the following methods in the Thread class are deprecated?

- A. `yield()`
- B. `stop()`
- C. `resume()`
- D. `suspend()`

You can use the \_\_\_\_\_ method(s) to temporarily release time for other threads.

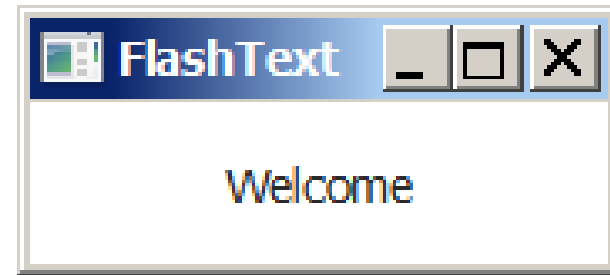
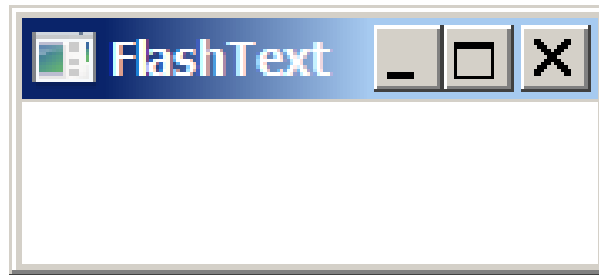
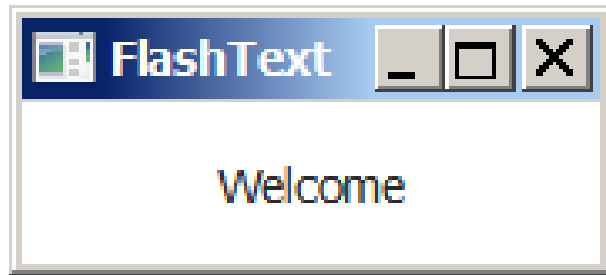
- A. `sleep(long milliseconds)`
- B. `yield()`
- C. `stop()`
- D. `suspend()`

# When you run the following program, what will happen?

```
public class Test extends Thread {  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.start();  
        t.start();  
    }  
  
    public void run() {  
        System.out.println("test");  
    }  
}
```

- A. Nothing is displayed.
- B. The program displays test twice.
- C. The program displays test once.
- D. An illegal java.lang.IllegalThreadStateException may be thrown because you just started the thread and the thread might have not yet finished before you start it again.

## 30.5 Case Study: Flashing Text (Optional)



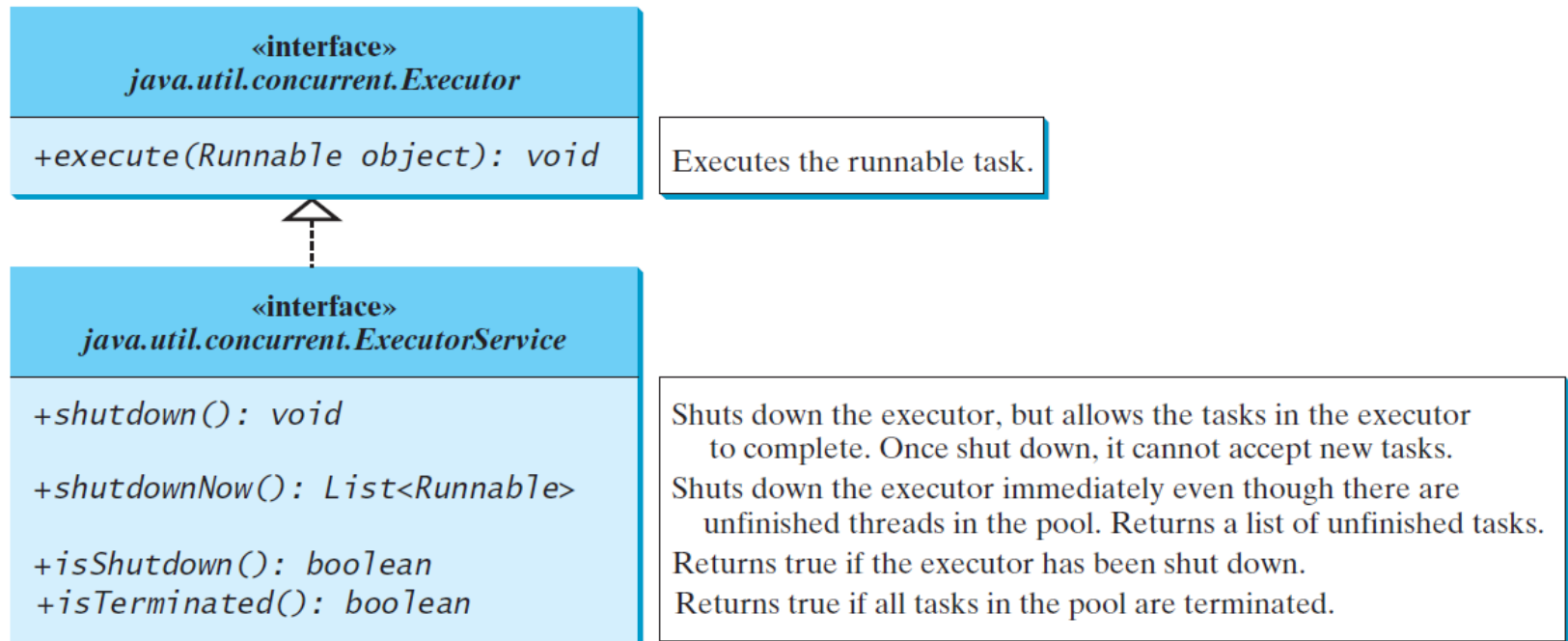
FlashText

## 30.6 Thread Pools

- Starting a new thread for each task could limit throughput and cause poor performance.
- 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.
- **ExecutorService** is a **subinterface** of **Executor**, as shown on the next page.

# The **Executor** interface and the **ExecutorService** subinterface

- The **Executor** **interface** *executes threads* and the **ExecutorService** **subinterface** *manages threads*.



# Creating Executors

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

## **java.util.concurrent.Executors**

+newFixedThreadPool(numberOfThreads:  
int): ExecutorService

+newCachedThreadPool():  
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.

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

ExecutorDemo

## 💡 Tip

- If you **need** to create a thread for just one task, use the **Thread class**.
- If you **need** to create threads for multiple tasks, it is better to **use a thread pool**.



Suppose there are three Runnable tasks, task1, task2, task3.  
How do you run them in a thread pool with 2 fixed threads?

- A. `new Thread(task1).start(); new Thread(task2).start(); new Thread(task3).start();`
- B. `ExecutorService executor = Executors.newFixedThreadPool(3);  
executor.execute(task1); executor.execute(task2); executor.execute(task3);`
- C. `ExecutorService executor = Executors.newFixedThreadPool(2);  
executor.execute(task1); executor.execute(task2); executor.execute(task3);`
- D. `ExecutorService executor = Executors.newFixedThreadPool(1);  
executor.execute(task1); executor.execute(task2); executor.execute(task3);`

# How do you create a cached thread pool?

- A. `ExecutorService executor = Executors.newCachedThreadPool();`
- B. `ExecutorService executor = Executors.newCachedThreadPool(1);`
- C. `ExecutorService executor = Executors.newCachedThreadPool(2);`
- D. `ExecutorService executor = Executors.newCachedThreadPool(3);`

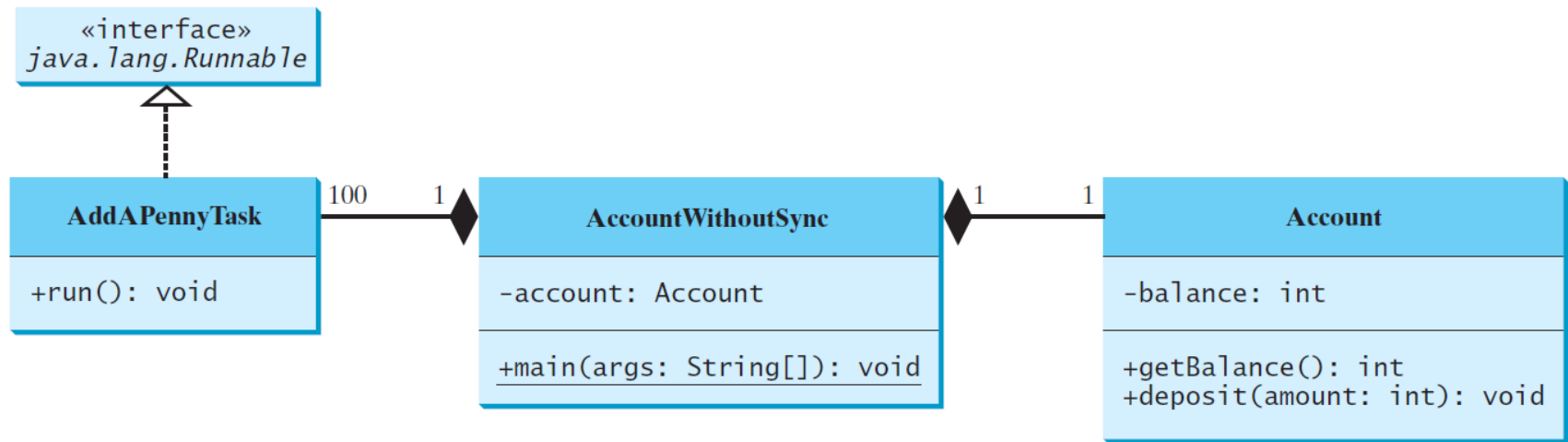
## 30.7 Thread Synchronization

- 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.



```

C:\book>java AccountWithoutSync
What is balance ? 5

C:\book>java AccountWithoutSync
What is balance ? 4

C:\book>java AccountWithoutSync
What is balance ? 7

C:\book>
  
```

AccountWithoutSync

# Race Condition

- What caused the error in the example? One 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.
- 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**.

## 30.7.1 The `synchronized` keyword

- To avoid race conditions, more than one thread must be prevented from simultaneously entering a certain part of the program, known as the *critical region*.
- The critical region in the Listing `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; one approach is to make `Account` thread-safe by adding the `synchronized` keyword in the deposit method in Line 38 as follows:

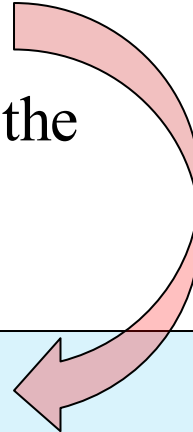
```
public synchronized void deposit(int amount)
```

# 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 (cont.)

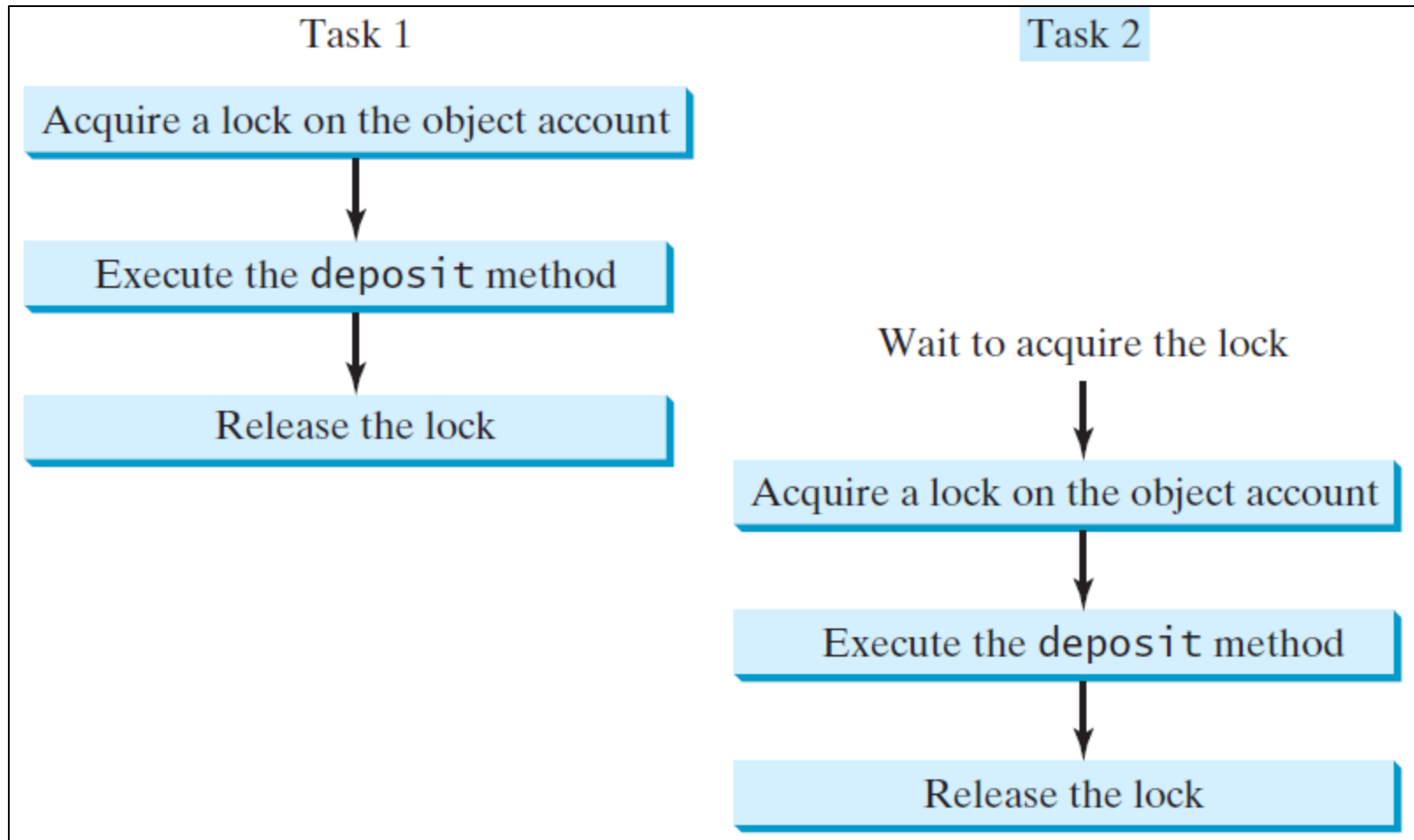
- With the **deposit method *synchronized***, the **preceding scenario cannot happen**.
- If **Task 2** starts to enter the method, and **Task 1** is already in the method, **Task 2** is **blocked until Task 1 finishes** the method.



Step	Balance	Task 1	Task 2
1	0	<code>newBalance = balance + 1;</code>	
2	0		<code>newBalance = balance + 1;</code>
3	1	<code>balance = newBalance;</code>	
4	1		<code>balance = newBalance;</code>



# Synchronizing Tasks



## 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;  
}
```

- 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 (cont.)

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

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

# 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:

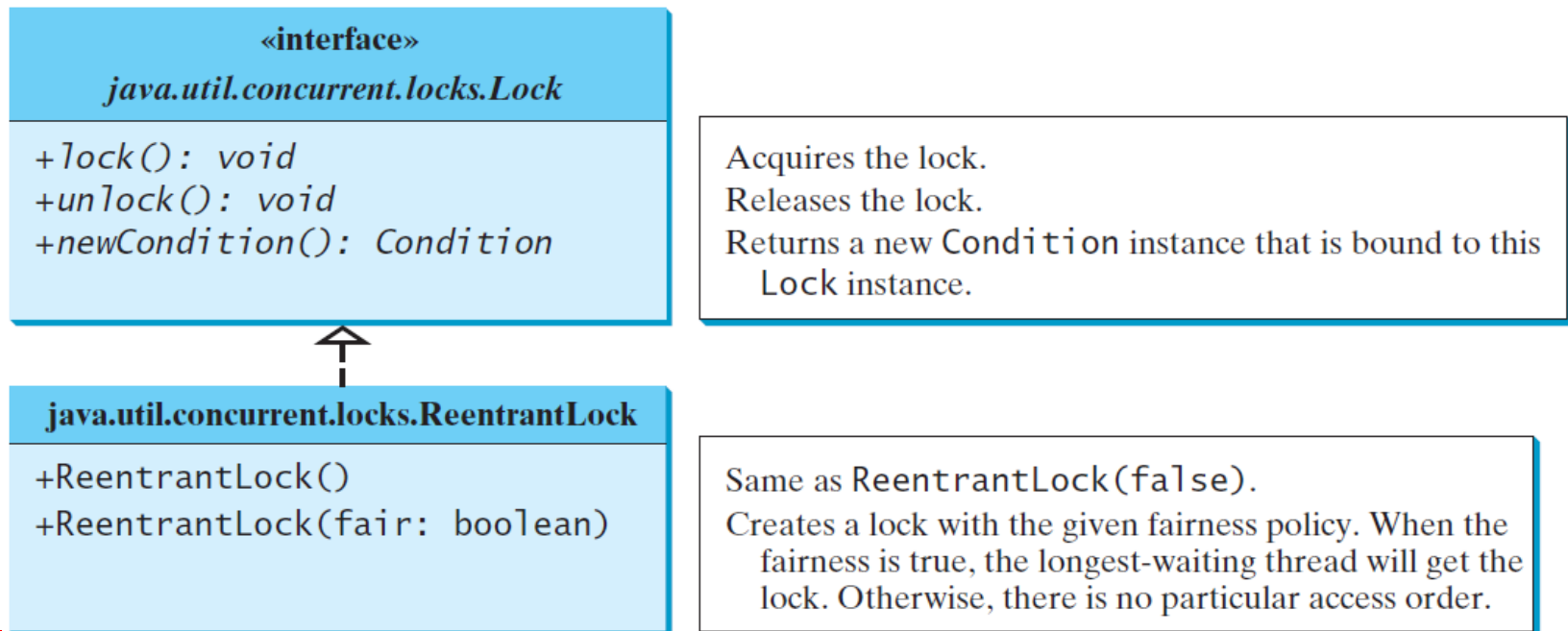
```
public void xMethod() {  
    synchronized (this) {  
        // method body  
    }  
}
```

The keyword to synchronize methods in Java is \_\_\_\_\_.

- A. synchronize
- B. synchronizing
- C. synchronized
- D. Synchronized

## 3.8 Synchronization Using Locks

- A **synchronized instance method** *implicitly* acquires a **lock** on the instance before it executes the method.
- Java enables you to **acquire locks** *explicitly*, which gives 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 mutually 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 **arbitrarily**.
- Programs using **fair** locks accessed by many threads may have poorer overall performance than those using the default setting, but have smaller variances in times to obtain locks and **prevent starvation**.

## Example: Using Locks

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

AccountWithSyncUsingLock



Which of the following are correct statements to create a Lock?

- A. `Lock lock = new Lock();`
- B. `Lock lock = new ReentrantLock();`
- C. `Lock lock = new ReentrantLock(true);`
- D. `Lock lock = new ReentrantLock(false);`

Which of the following are correct statements to create a Lock so the longest-wait thread will obtain the lock first?

- A. `Lock lock = new Lock();`
- B. `Lock lock = new ReentrantLock();`
- C. `Lock lock = new ReentrantLock(true);`
- D. `Lock lock = new ReentrantLock(false);`

You should always invoke the unlock method in the finally clause.

A. true

B. false

## 30.9 Cooperation Among Threads

- **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»

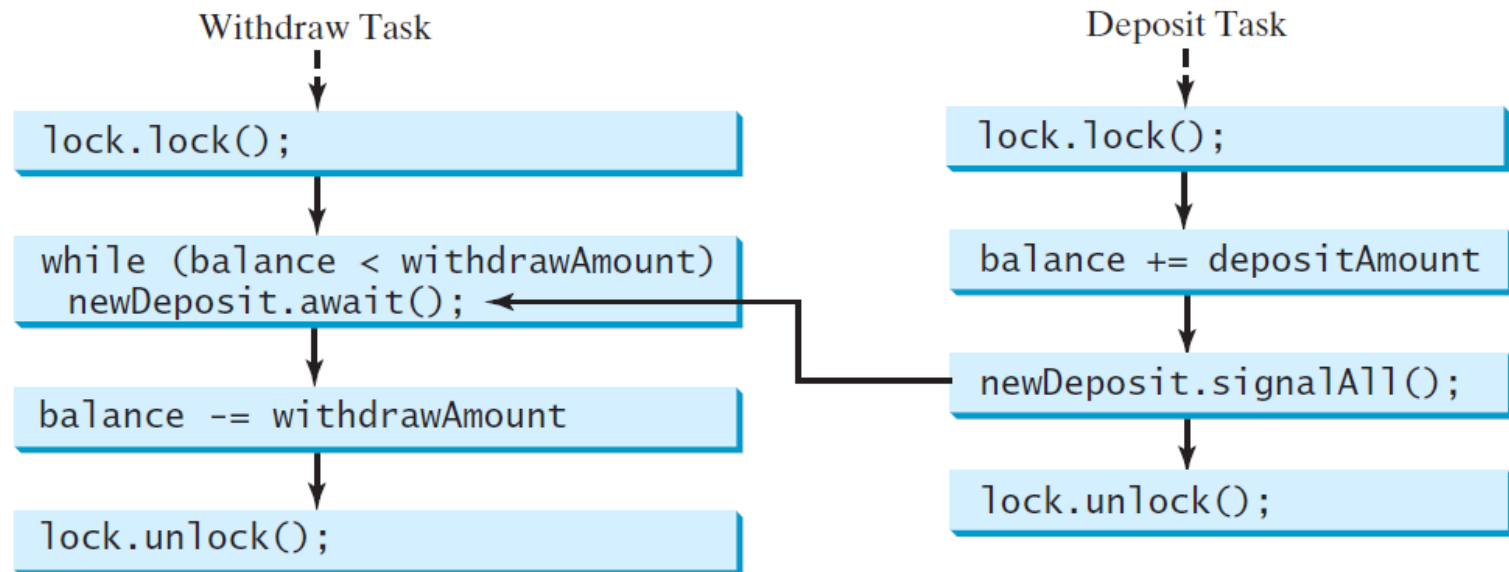
*java.util.concurrent.Condition*

```
+await(): void  
+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.

# Example: 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

- Objective: 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.

```

C:\book>java ThreadCooperation
Thread 1          Thread 2          Balance
Deposit 7         Withdraw 9         7
Deposit 1         Withdraw 4         8
Deposit 10        Withdraw 3         18
                  Withdraw 5         9
                  Withdraw 2         5
Deposit 9         Withdraw 2         2
                  Withdraw 5         11
Deposit 3         Withdraw 2         6
                  Withdraw 2         4
                  Withdraw 2         7

```

ThreadCooperation

# How do you create a condition on a lock?

- A. `Condition condition = lock.getCondition();`
- B. `Condition condition = lock.newCondition();`
- C. `Condition condition = Lock.newCondition();`
- D. `Condition condition = Lock.getCondition();`

Which method on a condition should you invoke to cause the current thread to wait until the condition is signaled?

- A. `condition.await();`
- B. `condition.wait();`
- C. `condition.waiting();`
- D. `condition.waited();`



Which method on a condition should you invoke to wake all waiting threads?

- A. `condition.wake();`
- B. `condition.signal();`
- C. `condition.wakeAll();`
- D. `condition.signalAll();`

## Java's Built-in Monitors (Optional)

Locks and conditions were new in Java 5. Prior to Java 5, thread communications were programmed using objects' 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.

# Java's Built-in Monitors (Optional, cont.)

## 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()` wakes up only one thread from a waiting queue.

# Example: Using Monitors (Optional)

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();
    }
}
```

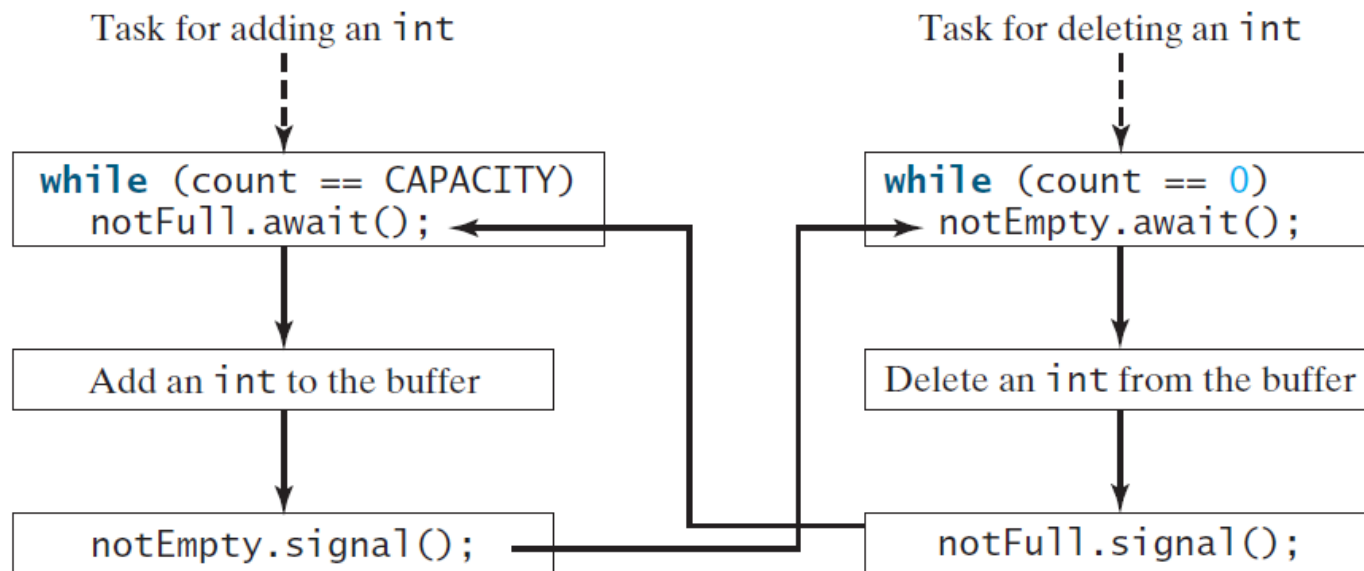
Task 2

```
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**.

## 30.10 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.



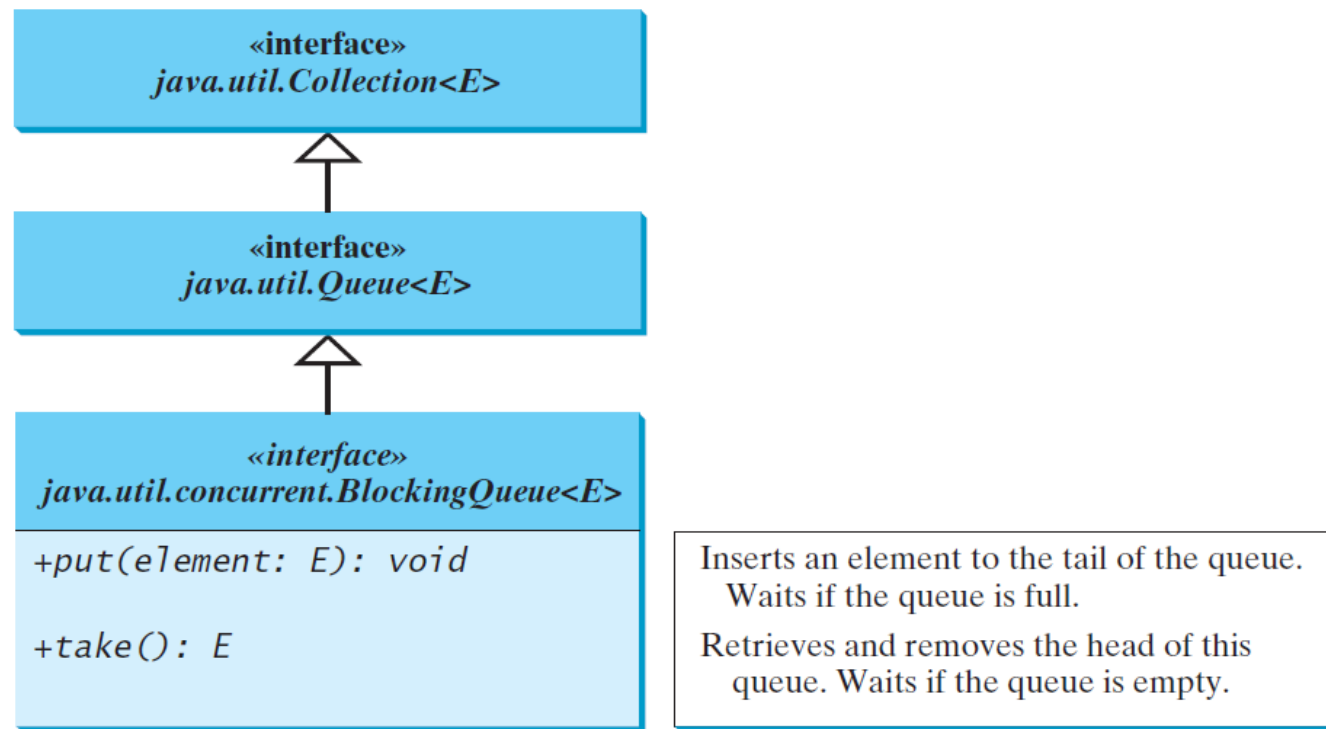
## 30.10 Case Study: Producer/Consumer (Optional)

- Listing 30.7 presents the complete program. The program contains the `Buffer` class (lines 50-101) and two tasks for repeatedly producing and consuming numbers to and from the buffer (lines 16-47). The `write(int)` method (lines 62-79) adds an integer to the buffer. The `read()` method (lines 81-100) deletes and returns an integer from the buffer.
- For simplicity, the buffer is implemented using a linked list (lines 52-53). Two conditions `notEmpty` and `notFull` on the lock are created in lines 59-60. 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

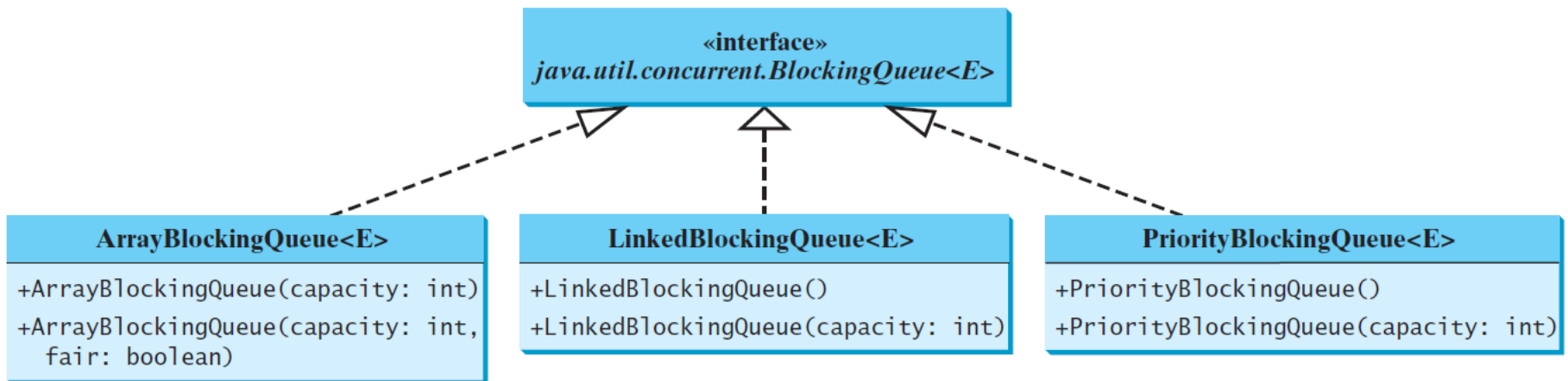
## 30.11 Blocking Queues (Optional)

- 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 (Optional)

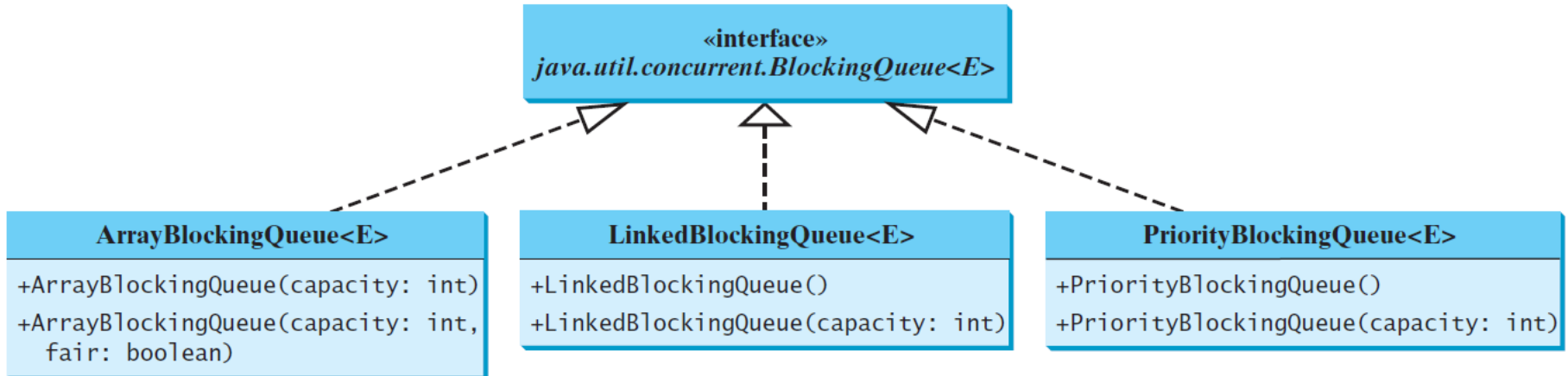
- Three concrete blocking queues **ArrayBlockingQueue**, **LinkedBlockingQueue**, and **PriorityBlockingQueue** are supported in Java.
- All are in the **java.util.concurrent** package.





## Concrete Blocking Queues (Optional)

- **ArrayBlockingQueue** implements a blocking queue using an array. You have to specify a capacity with an optional fairness policy 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.



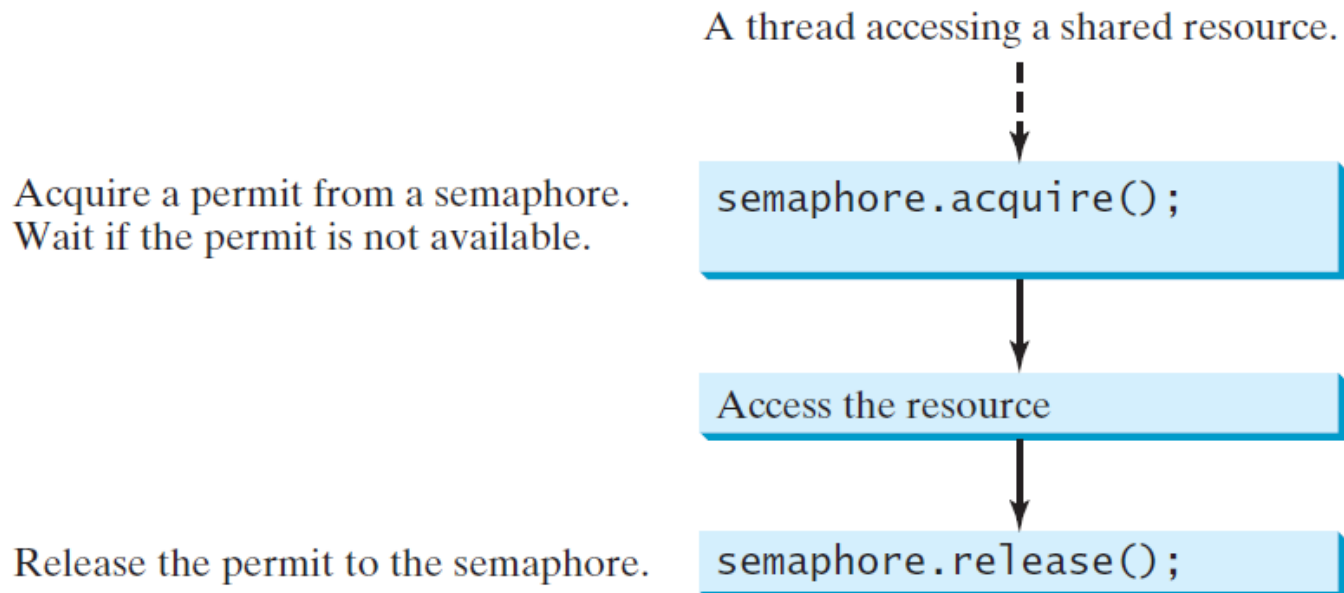
# Producer/Consumer Using Blocking Queues (Optional)

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

ConsumerProducerUsingBlockingQueue

## 30.12 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)
+Semaphore(numberOfPermits: int, fair:
  boolean)
+acquire(): void
+release(): void
```

Creates a semaphore with the specified number of permits. The fairness policy is false.

Creates a semaphore with the specified number of permits and the fairness policy.

Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.

Releases a permit back to the semaphore.

## Example: Semaphores

- A **semaphore** with just *one permit* can be used to **simulate** a *mutually exclusive lock*.
- The following listing revises the **Account** inner class using a **semaphore** to ensure that only one thread at a time can access the **deposit** method.

```
1 // An inner class for Account
2 private static class Account {
3     // Create a semaphore
4     private static Semaphore semaphore = new Semaphore(1);
5     private int balance = 0;

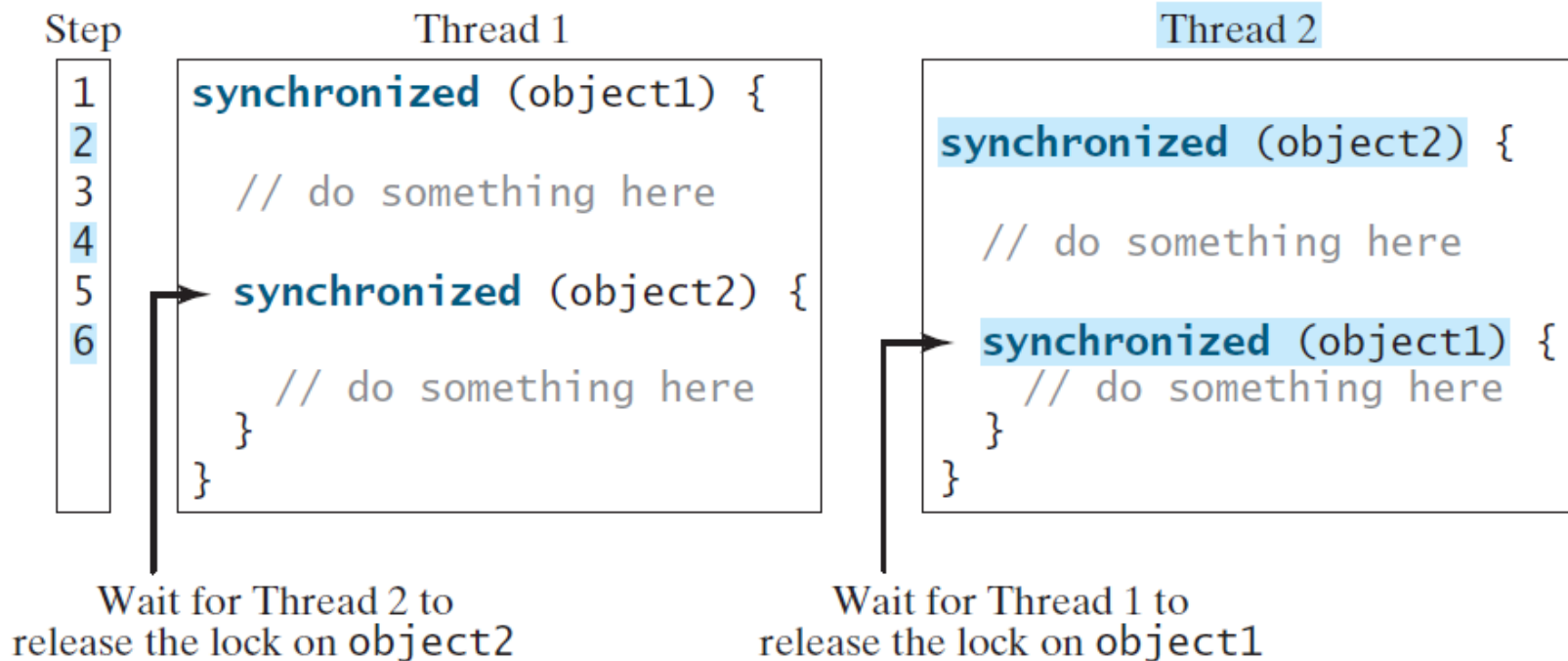
6
7     public int getBalance() {
8         return balance;
9     }
10
11 ( ... Continued on next page ...)
```

## Example: Semaphores (cont.)

```
11 public void deposit(int amount) {
12     try {
13         semaphore.acquire(); // Acquire a permit
14         int newBalance = balance + amount;
15
16         // This delay is deliberately added to magnify the
17         // data-corruption problem and make it easy to see
18         Thread.sleep(5);
19
20         balance = newBalance;
21     }
22     catch (InterruptedException ex) {
23     }
24     finally {
25         semaphore.release(); // Release a permit
26     }
27 }
28 }
```

## 30.13 Avoiding Deadlocks

- 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.



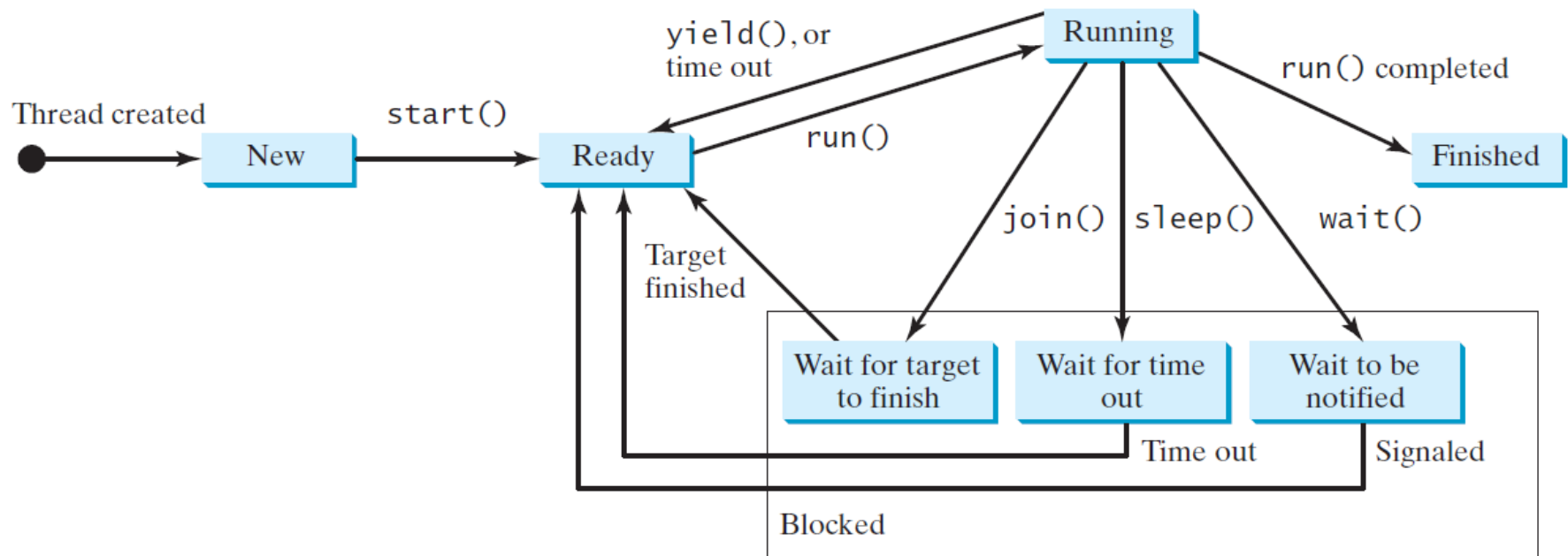
# Preventing Deadlock

- **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**.



## 30.14 Thread States

- A thread can be in one of five states: *New*, *Ready*, *Running*, *Blocked*, or *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.InterruptedExceptio`n is thrown.
- The `isInterrupted()` method tests whether the thread is interrupted.

## 30.15 Synchronized Collections (Optional)

- 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  
+synchronizedList(list: List): List  
+synchronizedMap(m: Map): Map  
+synchronizedSet(s: Set): Set  
+synchronizedSortedMap(s: SortedMap): SortedMap  
+synchronizedSortedSet(s: SortedSet): SortedSet
```

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.

## 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 **java.util.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.

# 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`.

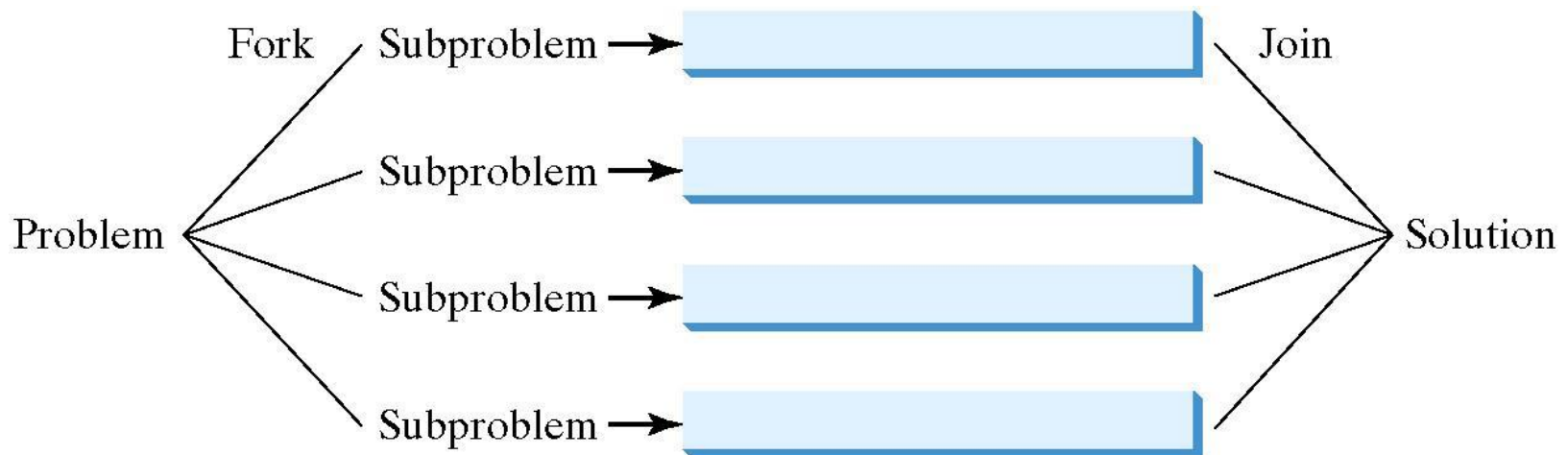
# 30.16 Parallel Programming

## 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 introduced the new ***Fork/Join Framework*** for **parallel programming**, which utilizes the **multicore processors**.

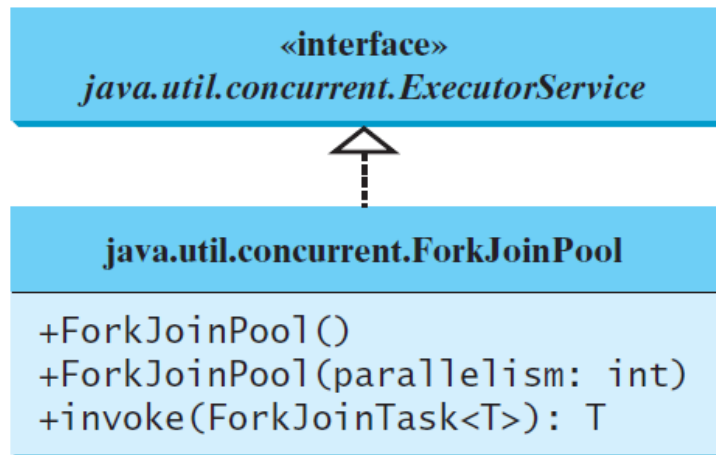
# The Fork/Join Framework

- 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**.

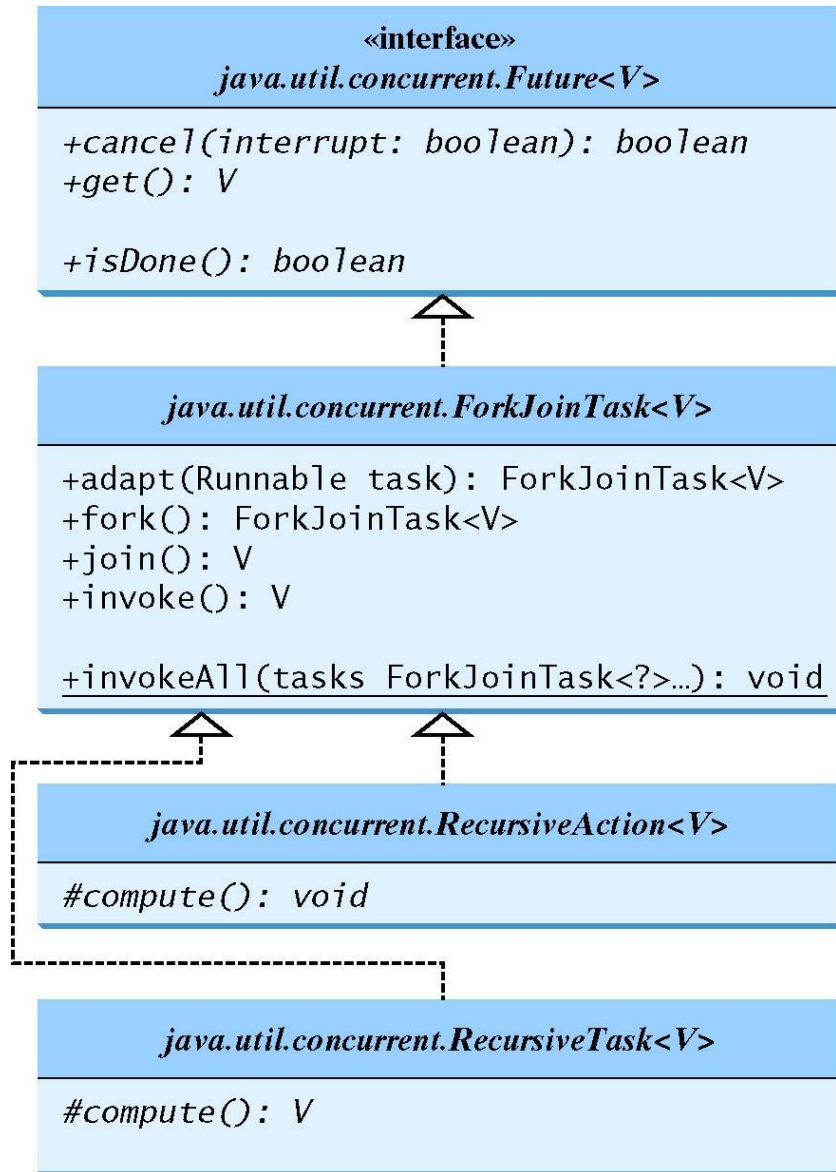


See Figure 30.7

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



Attempts to cancel this task.

Waits if needed for the computation to complete and returns the result.

Returns true if this task is completed.

Returns a `ForkJoinTask` from a runnable task.

Arranges asynchronous execution of the task.

Returns the result of computations when it is done.

Performs the task and awaits for its completion, and returns its result.

Forks the given tasks and returns when all tasks are completed.

Defines how task is performed.

Defines how task is performed. Return the value after the task is completed.

- `ForkJoinTask` is the **abstract base class** for tasks.
- A `ForkJoinTask` is a *thread-like entity*, but it is **much lighter than a normal thread** because **huge numbers** of tasks and subtasks can be **executed by** a small number of **actual threads** in a `ForkJoinPool`.
- The **tasks** are **primarily coordinated** using `fork()` and `join()`.
  - Invoking `fork()` on a task **arranges asynchronous execution**, and invoking `join()` **waits until the task is completed**.
  - The `invoke()` and `invokeAll(tasks)` methods **implicitly invoke `fork()`** to execute the task **and `join()`** to wait for the tasks to complete, and return the result, if any.
  - Note that the **static method `invokeAll`** takes a **variable number of `ForkJoinTask` arguments** using the **`...` syntax**.

- The **Fork/Join Framework** is designed to **parallelize divide-and-conquer solutions**, which are **naturally recursive**.
- **RecursiveAction** and **RecursiveTask** are two **subclasses** of **ForkJoinTask** .
- To define a **concrete task class**, your class **should extend** **RecursiveAction** or **RecursiveTask**.
- **RecursiveAction** is for a task that **doesn't** return a value, and **RecursiveTask** is for a task that **does** return a value.
- Your **task class** should **override** the **compute()** method to **specify how** a task is performed.

## Example

- The following program gives a *parallel* implementation of the *merge sort* algorithm (introduced in Section 23.4) and compares its execution time with a sequential sort.

ParallelMergeSort

- In general, a problem can be **solved in *parallel*** using the following **pattern**:

```
if (the program is small)
    solve it sequentially;
else {
    divide the problem into nonoverlapping subproblems;
    solve the subproblems concurrently;
    combine the results from subproblems to solve the
        whole problem;
}
```

## Example

- The following program develops a *parallel* method that **finds** the **maximal number** in a list.

ParallelMax