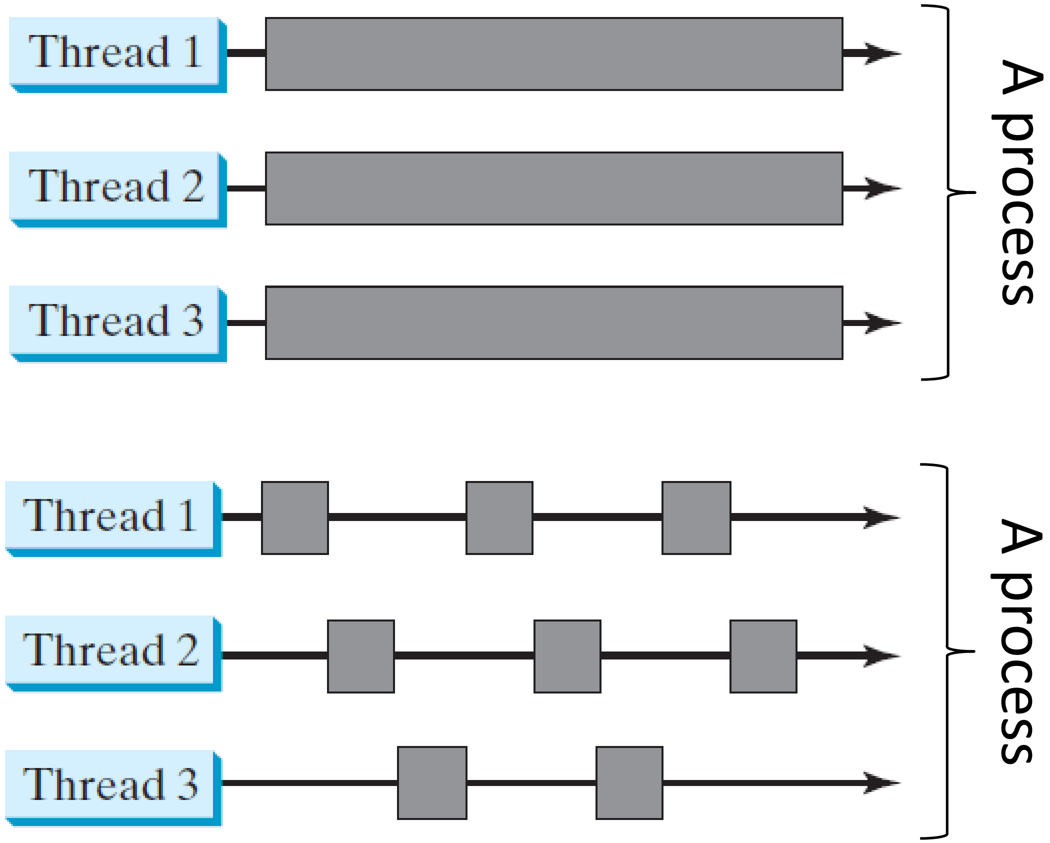
Multithreaded Programming

Shing-Chi Cheung
Computer Science and Engineering
HKUST

Process and Threads

Multiple threads on multiple CPU cores

Multiple threads sharing a single CPU core

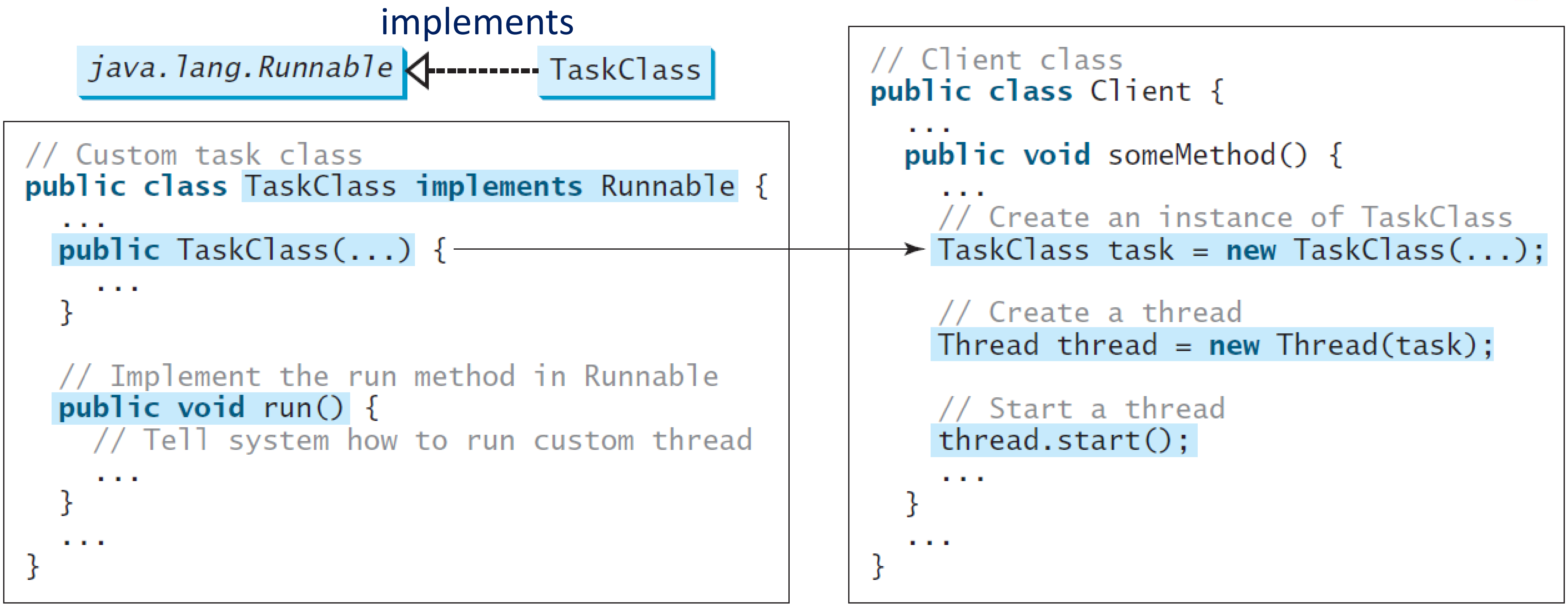


A process is a program in execution, which may consist of multiple concurrent execution units known as threads.

Threads in a process share the same memory heap space.

Threads need to be synchronized if they have dependency.

Creating Tasks and Threads (6 important steps)



```
// 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)

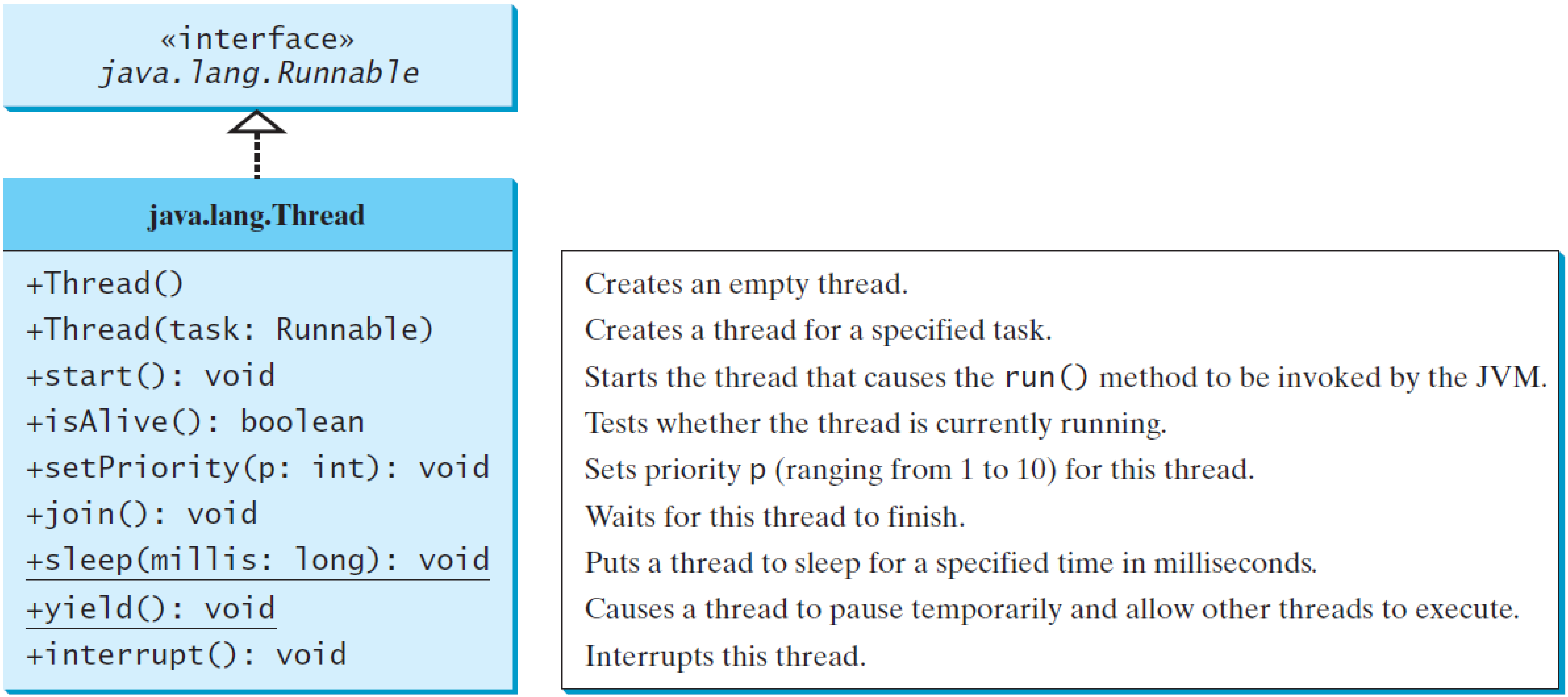
Example: Using the Runnable Interface to Create and Launch Threads

- Objective: Create and run three threads:
  - ❑ The first thread prints ‘a’ 20 times
  - ❑ The second thread prints ‘b’ 20 times
  - ❑ The third thread prints ‘c’ 20 times

```java
public void run() {
    for (var i = 0; i < times; i++) {
        System.out.print(charToPrint);
    }
}
```

TaskThreadDemo.java

The Thread Class



«interface»
java.lang.Runnable

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.

The Static yield() Method

- We can use the yield() method to release time for other threads. For example, we can modify the run method as shown
- The thread yields every time after a character is printed. So, the thread runs relatively slower

```java
public void run() {
    for (var i = 0; i < times; i++) {
        System.out.print(charToPrint);
        Thread.yield();
    }
}
```

TaskThreadYieldDemo.java

The Static sleep(milliseconds) Method

- The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, we can modify the run method as shown
- The thread sleeps 100 milliseconds every time after a character is printed

```
public void run() {
  try {
    for (var i = 0; i < times; i++) {
      System.out.print(charToPrint);
      // sleep 100 milliseconds
      Thread.sleep(100);
    }
  } catch (InterruptedException ex) {
  }
}
```

TaskThreadSleepDemo.java

The join() Method

- We can use the join() method to force one thread to wait for another thread to finish. For example, the run method can be modified as shown
- After printing 10 characters, the thread pauses until completion of thread4

```
public void run() {
    var thread4 = new Thread(
        new PrintChar('d', 20));
    thread4.start();
    try {
        for (var i = 0; i < times; i++) {
            System.out.print(charToPrint);
            if (i >= 10) thread4.join();
        }
    } catch (InterruptedException ex) {
    }
}
```

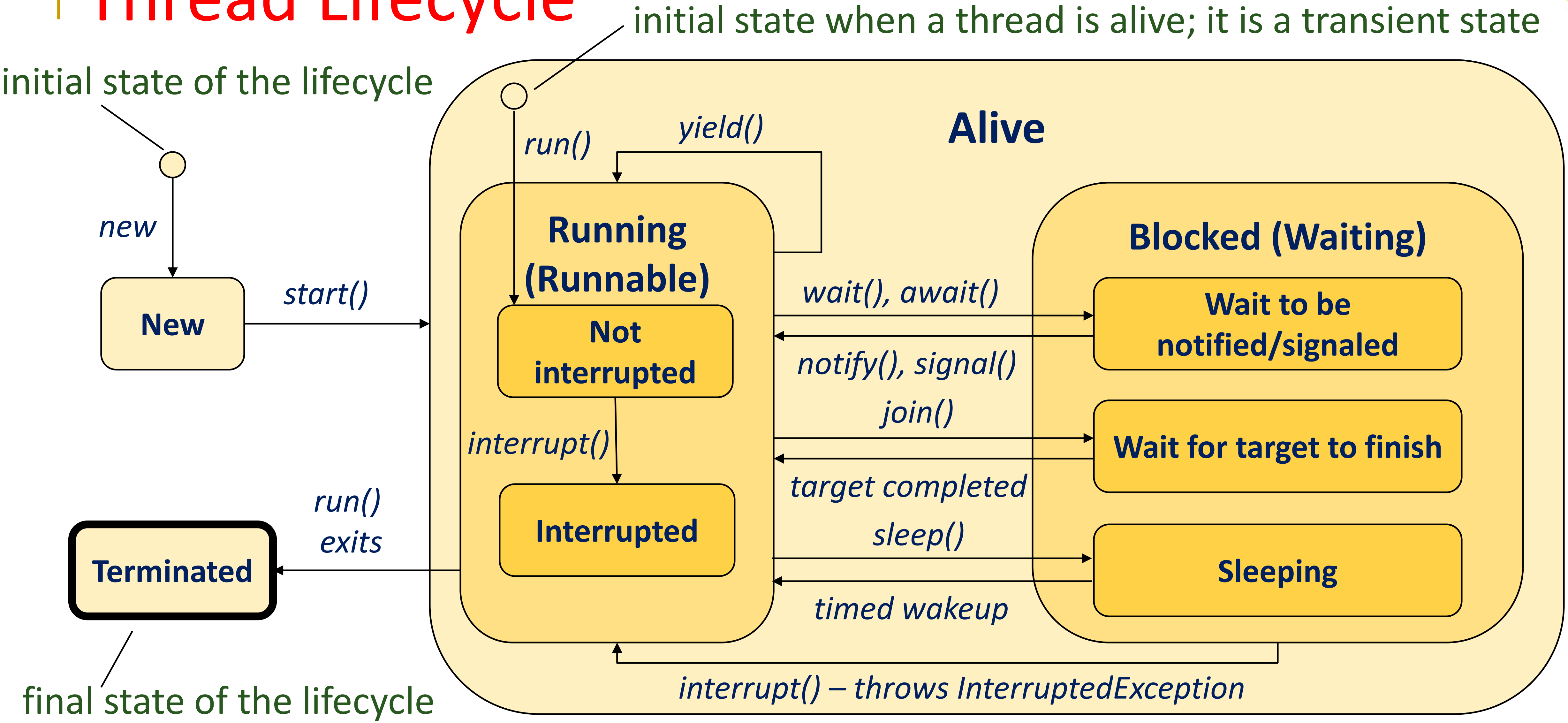
TaskThreadJoinDemo.java

Thread States

A thread can be in one of the following four major states

- New: thread is created but not yet started
- Running: thread is either being executed or to be scheduled for execution
- Blocked: thread is either blocked by monitor locks (to be discussed soon) or awaiting some events for a specified waiting time
- Terminated: thread exited

Thread Lifecycle



initial state when a thread is alive; it is a transient state
initial state of the lifecycle
new
New
start()
Alive
run()
yield()
Running (Runnable)
Not interrupted
interrupt()
Interrupted
Blocked (Waiting)
wait(), await()
Wait to be notified/signaled
notify(), signal()
join()
Wait for target to finish
target completed
sleep()
Sleeping
timed wakeup
interrupt() – throws InterruptedException
run() exits
Terminated
final state of the lifecycle

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

- isAlive() method returns true if the thread has started but not yet terminated; otherwise false
- interrupt() method interrupts a thread in the following way:
  - If the thread is in Running state, its interrupted flag is set
  - If the thread is in Blocked state; it enters Running state and throws java.io.InterruptedException
- The isInterrupt() method tests whether the current thread's interrupted flag has been set

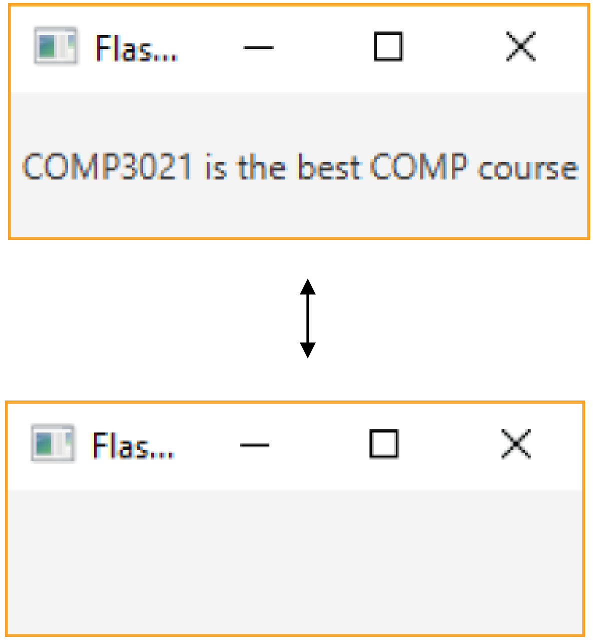
The deprecated stop(), suspend(), and resume() Methods

- The Thread class also supports the stop(), suspend(), and resume() methods.
- As of Java 2, these methods are deprecated (or outdated) because they are known to be inherently unsafe.
- We should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.
- Advice: Never use these three methods

Thread Priority

- Each thread is assigned a default priority of Thread.NORM_PRIORITY
  - We 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



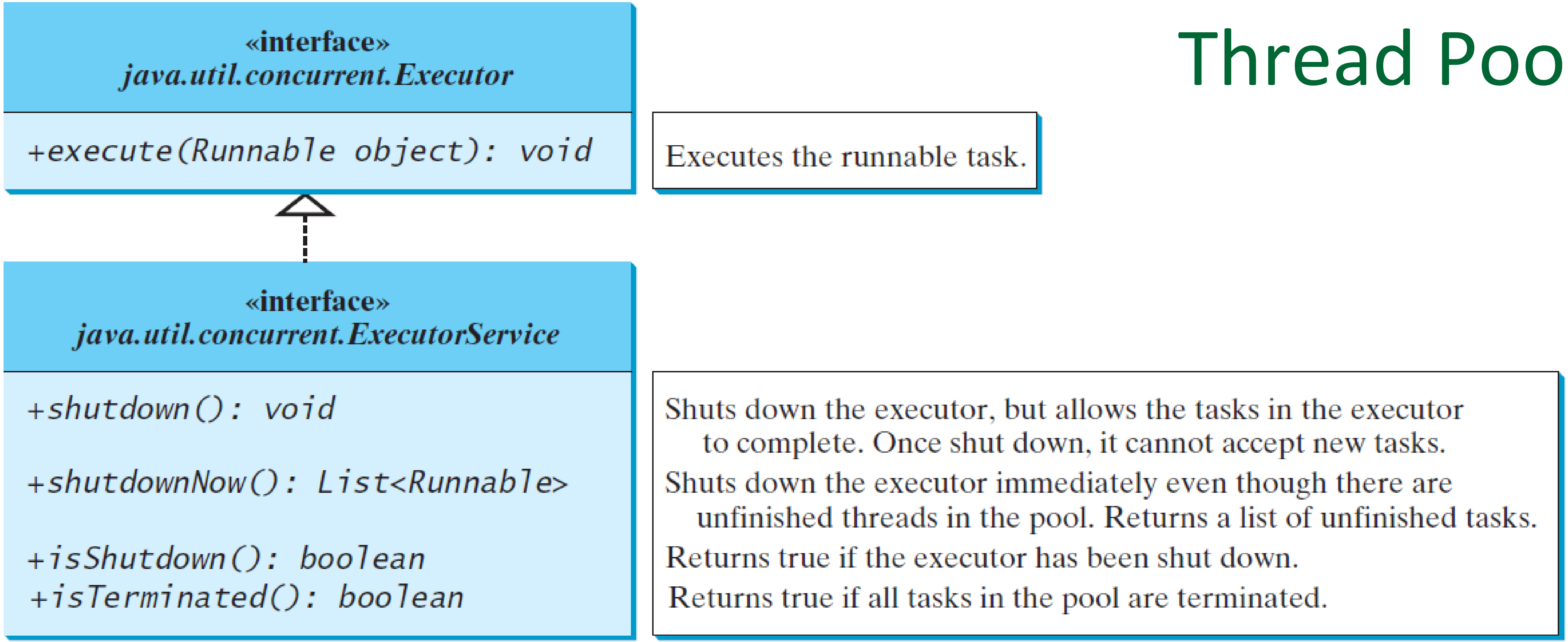
COMP3021 is the best COMP course

```java
public void start(Stage primaryStage) {
  var pane = new StackPane();
  var lblText = new Label("COMP3021 is the best COMP course");
  pane.getChildren().add(lblText);
  new Thread( () -> {
    try {
      while (true) {
        if ( lblText.getText().trim().length() == 0 )
          text = "COMP3021 is the best COMP course";
        else
          text = "";
        Platform.runLater( () -> lblText.setText(text) );
        Thread.sleep(200);
      }
    }
    catch (InterruptedException ex) {
    }
  } ).start();
  …
```

Implementation of run()

FlashText.java

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.

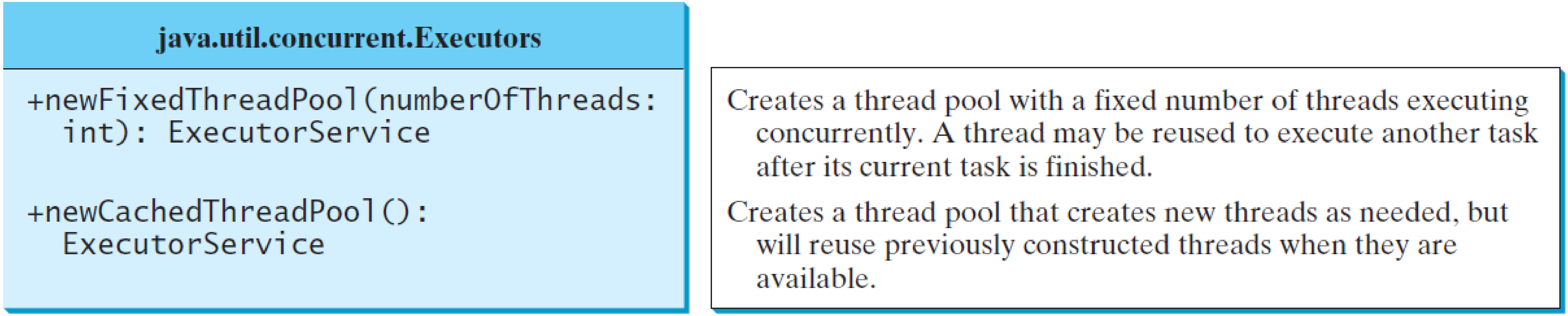
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shut down.

Returns true if all tasks in the pool are terminated.

- Starting a new thread for each task limits throughput and causes poor performance. A thread pool is ideal to manage the number of tasks executing concurrently. Use the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks. ExecutorService is a subinterface of Executor.

Creating Executors



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.

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

ExecutorDemo.java

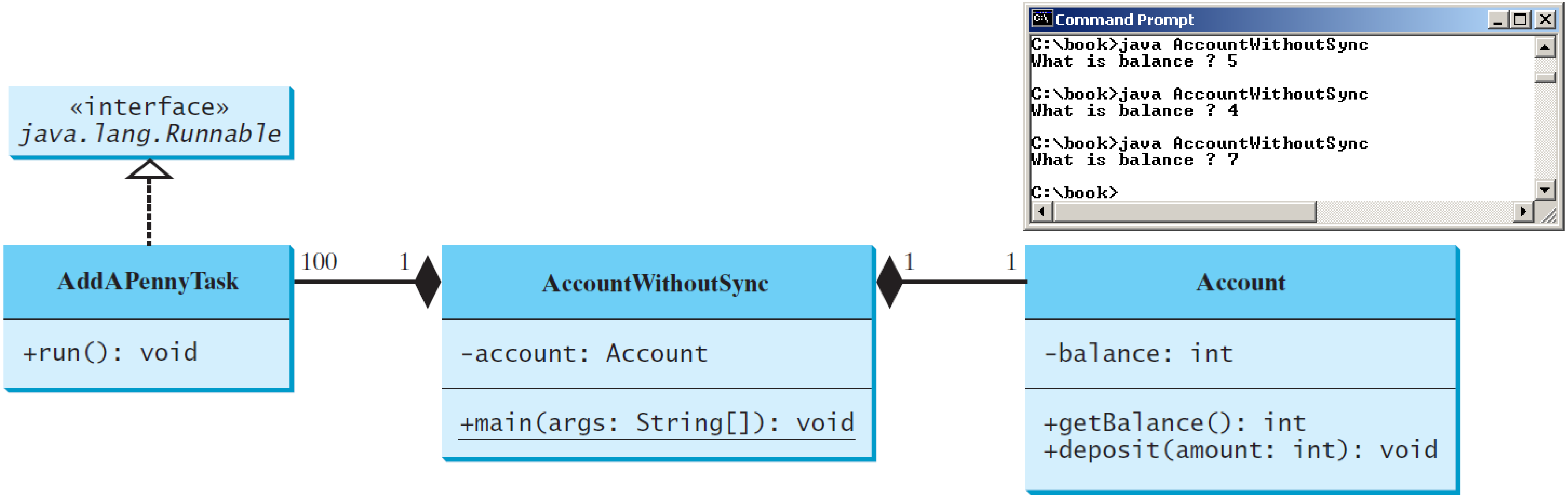
Thread Synchronization

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);

- 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.

Example: Showing Resource Conflict

- We create and launch one hundred threads, each of which adds a penny to an account that 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>
```

A scenario where conflicts arise

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 overwrites Task 1's result. The problem is that Task 1 and Task 2 are accessing a common resource in a way that causes conflicts
- This is a problem known as *race conditions* in multithreaded programs
- A class is said to be *thread-safe* if instances of the class are free from race conditions in the presence of multiple threads. Account class in the preceding example is not thread-safe

AccountWithoutSync.java

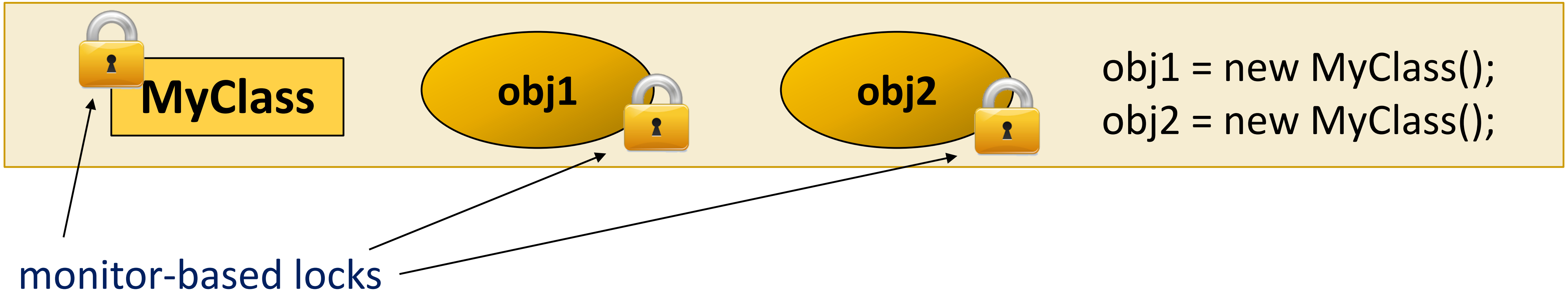
The synchronized keyword

- To avoid race conditions, only one thread is allowed to access certain part of the program, known as critical region
- The critical region is the entire deposit method. We can use the synchronized keyword to declare the entire method to be a critical region
- To correct the previous race problem, we can make Account thread-safe by adding the synchronized keyword in the deposit method:

```
public synchronized void deposit(double amount) { … }
```

AccountSync.java

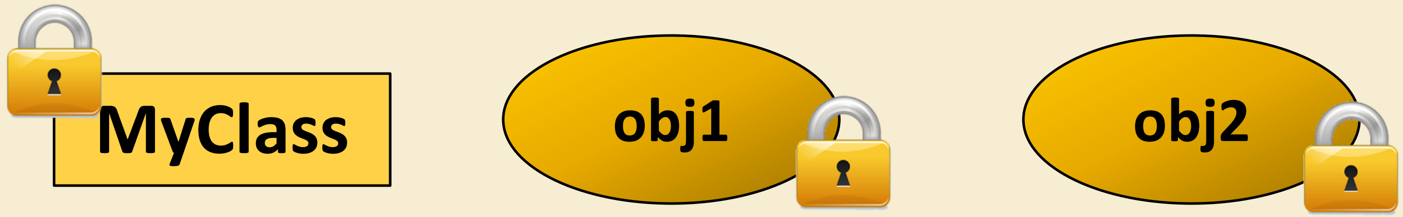
Thread Synchronization using Monitor-based Locks



```
obj1 = new MyClass();
obj2 = new MyClass();
```

- JVM assigns a separate lock to each class and each (object) instance
  - We call them **monitor-based locks** or **implicit locks**

Synchronization using Implicit (Monitor-based) Lock



```
obj1 = new MyClass();
obj2 = new MyClass();
```

1. A thread needs to acquire a lock before it executes a synchronized method
   - For a synchronized instance method, it acquires the lock of the associated instance
   - For a synchronized static method, it acquires the lock of the associated class
2. If the thread successfully acquires the lock, it executes the synchronized method and releases the lock when it finishes the method. The acquisition and release are performed implicitly. No extra method calls are needed.
3. If the thread fails to acquire the lock, its execution is blocked
   - It repeats step 1 when the lock is released by another thread

Thread Synchronization using Monitor-based Locks

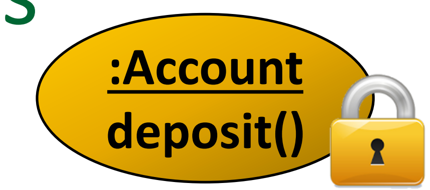
Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

- With the deposit method synchronized, the above racing condition cannot happen
- If Task 2 starts to execute the deposit method before Task 1 finishes the method, Task 2 is blocked

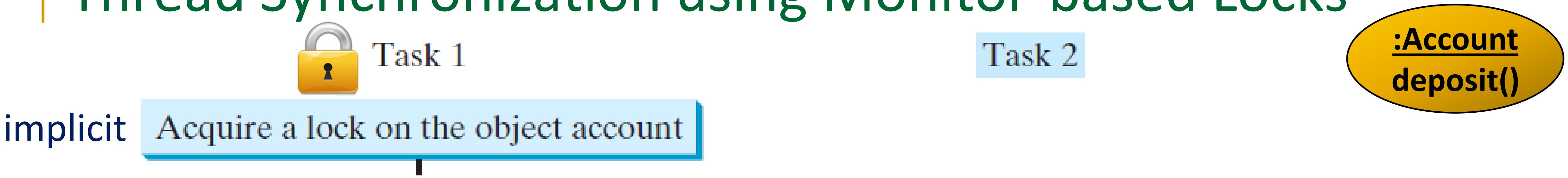
Thread Synchronization using Monitor-based Locks

Task 1

Task 2



Thread Synchronization using Monitor-based Locks



Task 1

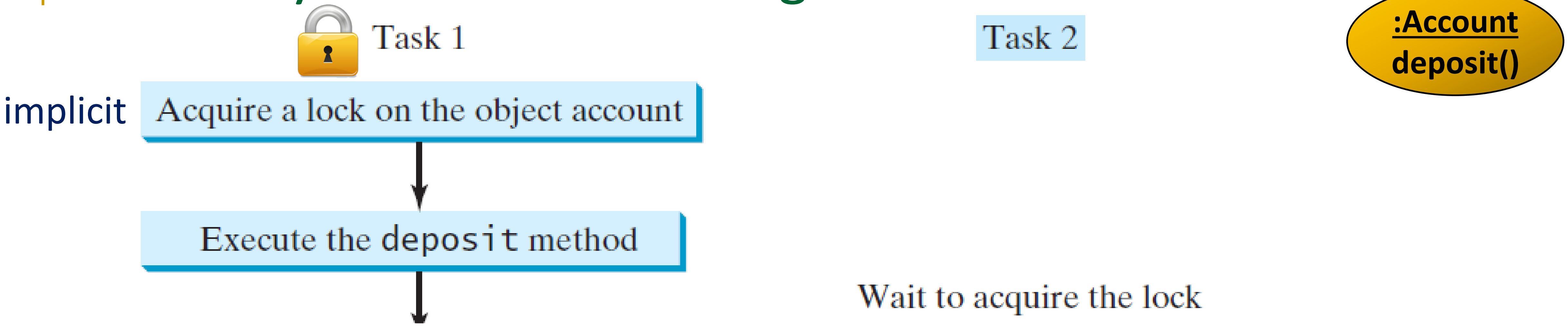
Task 2

:Account
deposit()

implicit

Acquire a lock on the object account

Thread Synchronization using Monitor-based Locks



Task 1

Task 2

:Account

deposit()

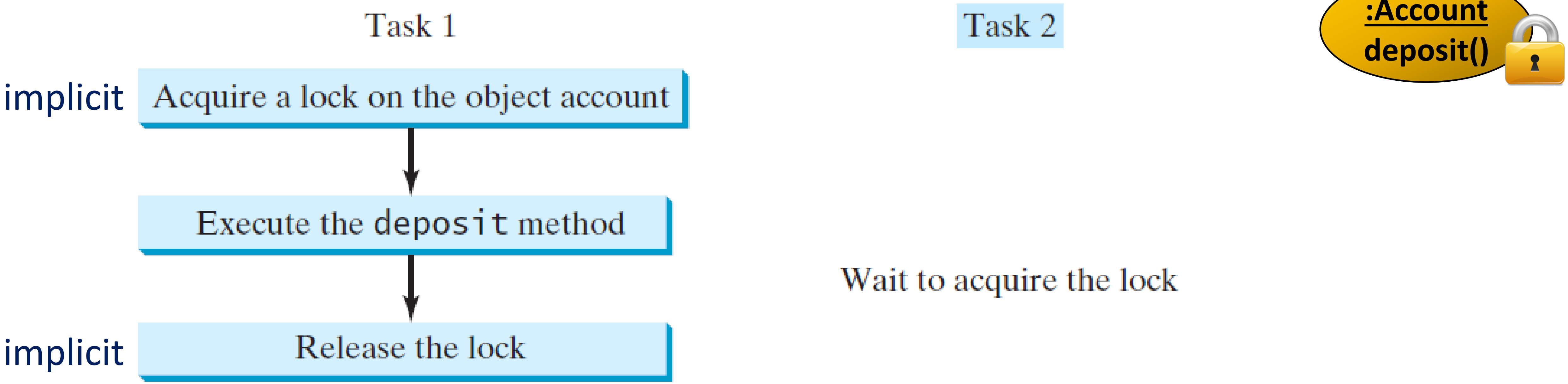
implicit

Acquire a lock on the object account

Execute the `deposit` method

Wait to acquire the lock

Thread Synchronization using Monitor-based Locks



:Account
deposit()

Task 1

Task 2

implicit

Acquire a lock on the object account

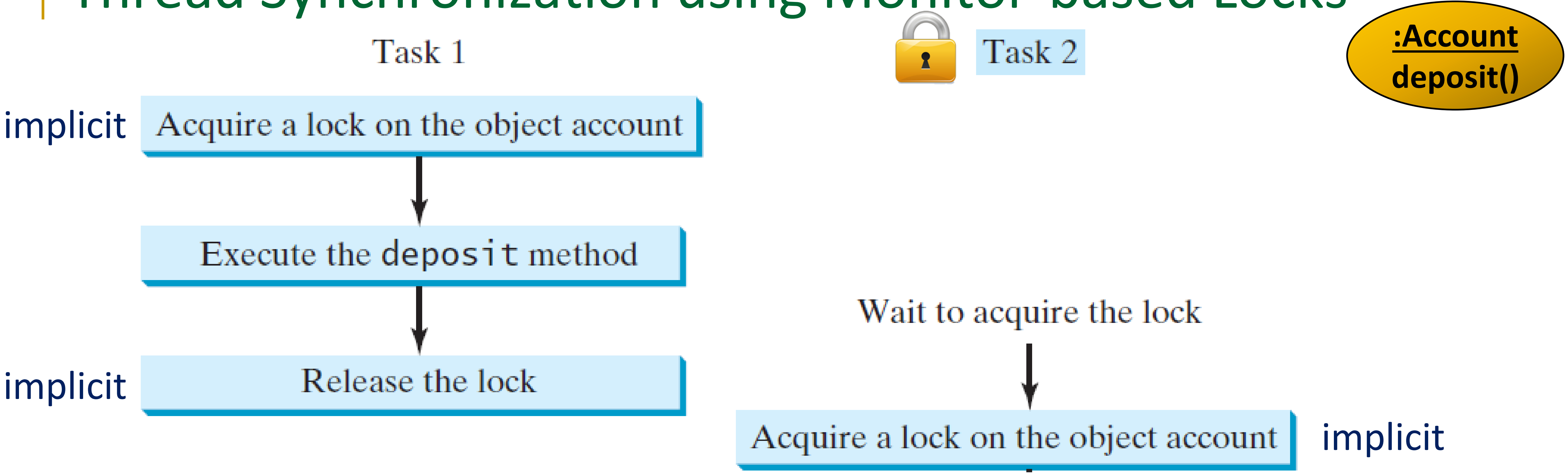
Execute the deposit method

implicit

Release the lock

Wait to acquire the lock

Thread Synchronization using Monitor-based Locks



:Account
deposit()

Task 1

implicit — Acquire a lock on the object account

Execute the deposit method

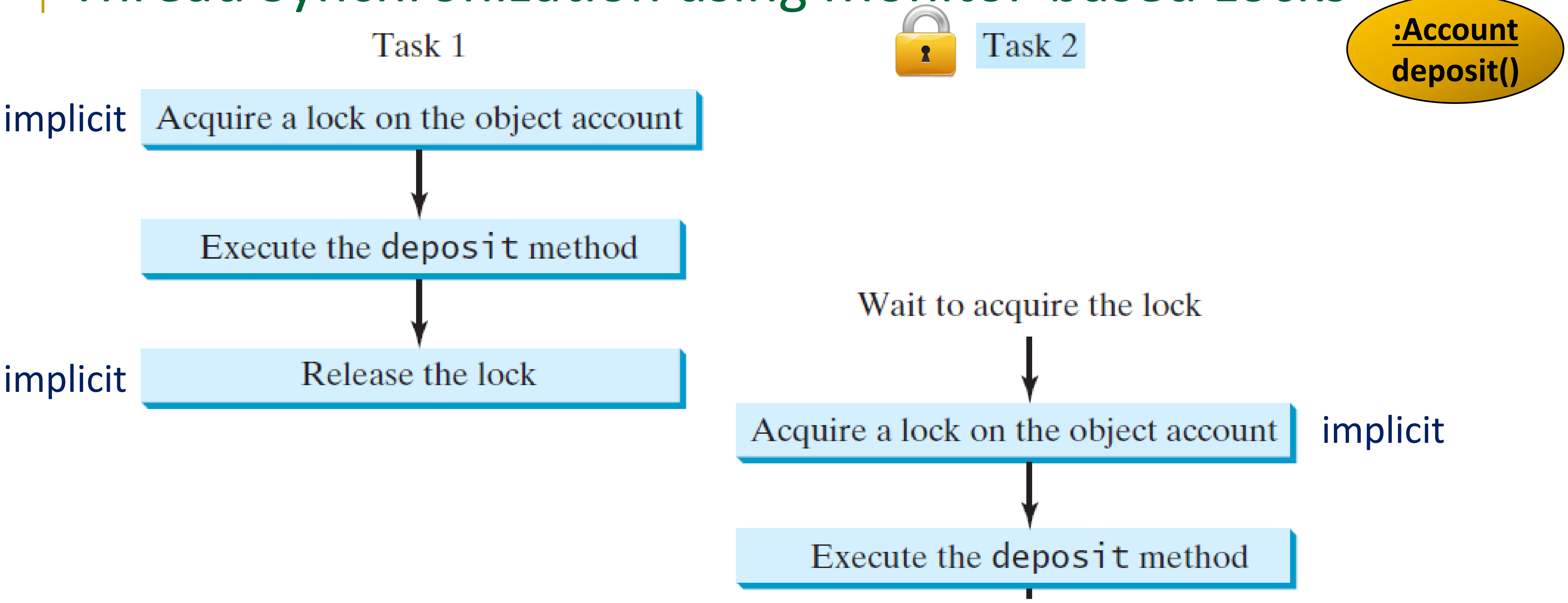
implicit — Release the lock

Task 2

Wait to acquire the lock

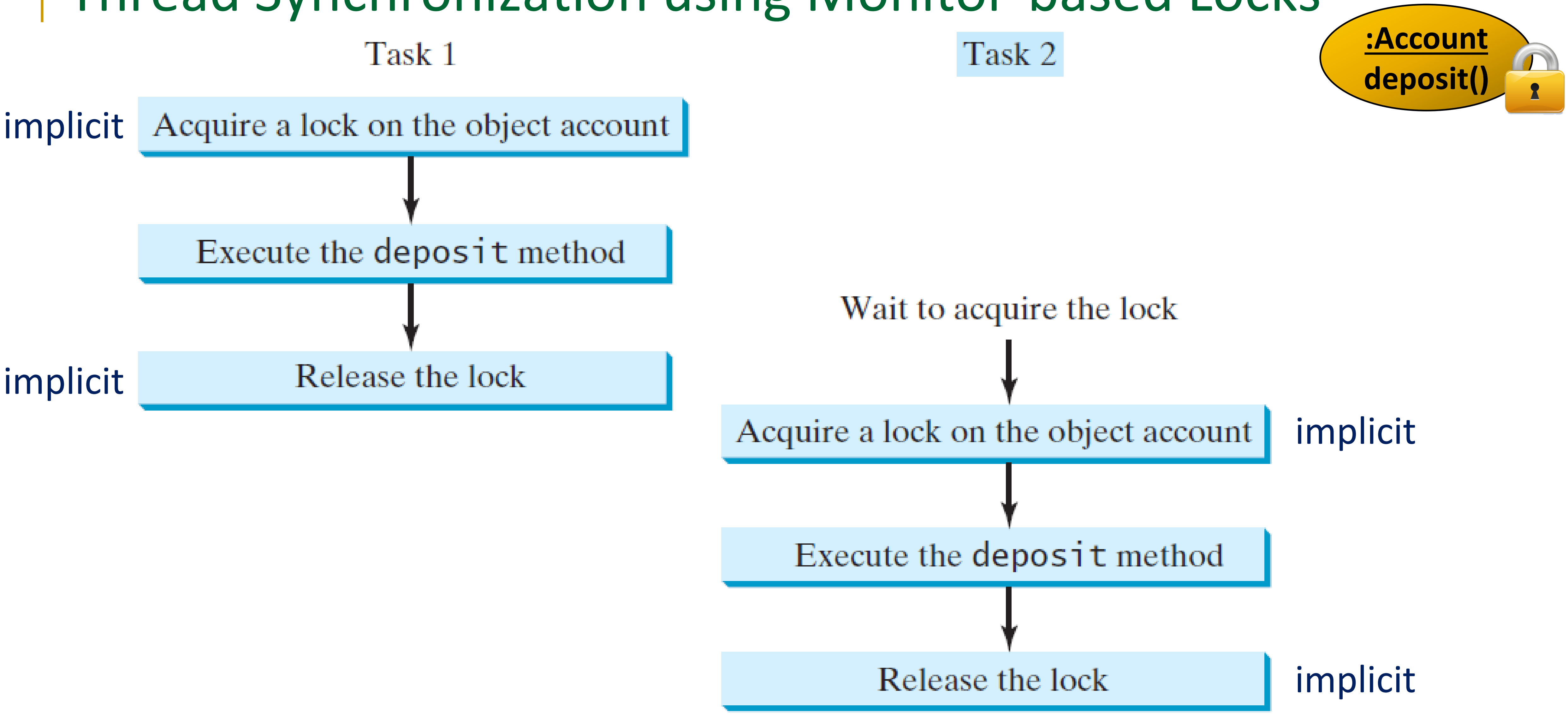
Acquire a lock on the object account — implicit

Thread Synchronization using Monitor-based Locks



:Account
deposit()
Task 1
implicit
Acquire a lock on the object account
Execute the deposit method
implicit
Release the lock
Task 2
Wait to acquire the lock
Acquire a lock on the object account
implicit
Execute the deposit method

Thread Synchronization using Monitor-based Locks



:Account
deposit()

Task 1

implicit Acquire a lock on the object account

Execute the `deposit` method

implicit Release the lock

Task 2

Wait to acquire the lock

Acquire a lock on the object account implicit

Execute the `deposit` method

Release the lock implicit

Thread Synchronization using Synchronized Statements

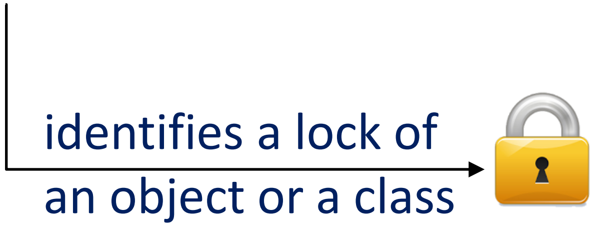
```
synchronized (expr) { // expr must be evaluated to an object/class reference
    statements;
}
```

identifies a lock of an object or a class

- A synchronized statement can be used to acquire a lock on a specific object, not limited to this object, when executing a block of the code in a method. This block is referred to as a synchronized block
  - ❑ synchronized (this) { … }   // synchronized using the current instance
  - ❑ synchronized (new Account()) { … }   // synchronized using a new instance
  - ❑ synchronized (Account.class) { … }   // synchronized using a class

Thread Synchronization using Synchronized Statements

```
synchronized (expr) { // expr must be evaluated to object/class reference
    statements;
}
```



identifies a lock of an object or a class

- The expression expr must be evaluated to an object reference or a class reference. If the object or class is already locked by another thread, the thread is blocked until the lock is successfully acquired
- When the lock is acquired, the statements in the synchronized block are executed, and then the lock is released after these statements have been executed.

AccountWithSyncBlock.java

Synchronized Statements vs. Synchronized Methods

- Any synchronized instance method can be converted into an instance method with a synchronized block

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

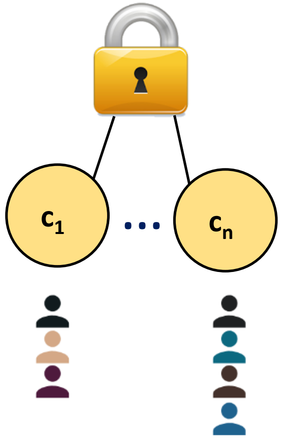
=

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

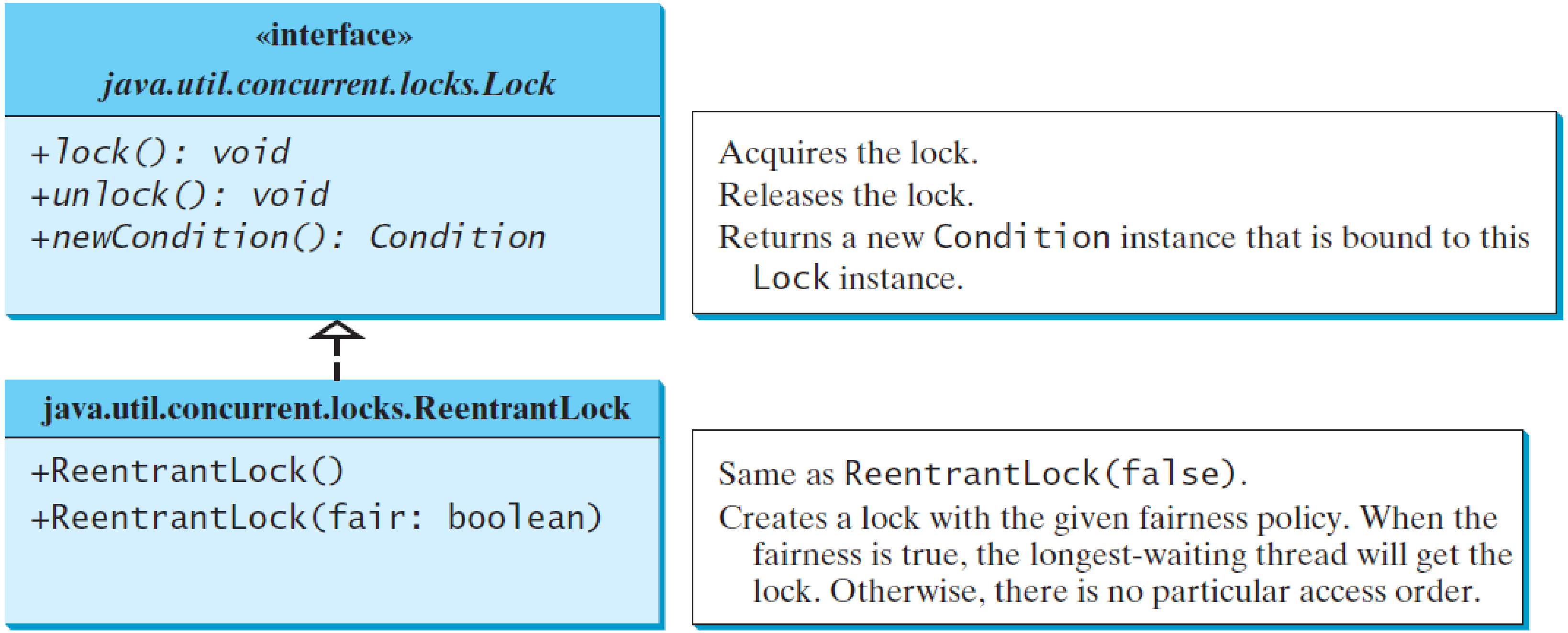
Q: Why synchronized statements?

A: Synchronized statements enable us to synchronize threads at statement granularity instead of method granularity

Thread Synchronization Using Explicit Locks



Queues of waiting threads



«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

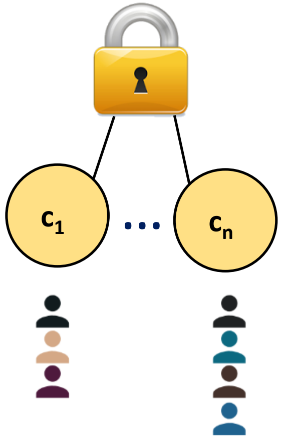
```
+ReentrantLock()
+ReentrantLock(fair: boolean)
```

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.

Need to call these methods explicitly

- Besides using monitor-based locks, we can synchronize threads by explicit locks
- An explicit lock is an instance of the Lock interface, which declares the methods to acquire and release locks. A lock may use the newCondition() method to create any number of Condition objects, which can be used for thread communications

Thread Synchronization Using Explicit Locks



Queues of waiting threads

Example: A lock of a n-slot buffer

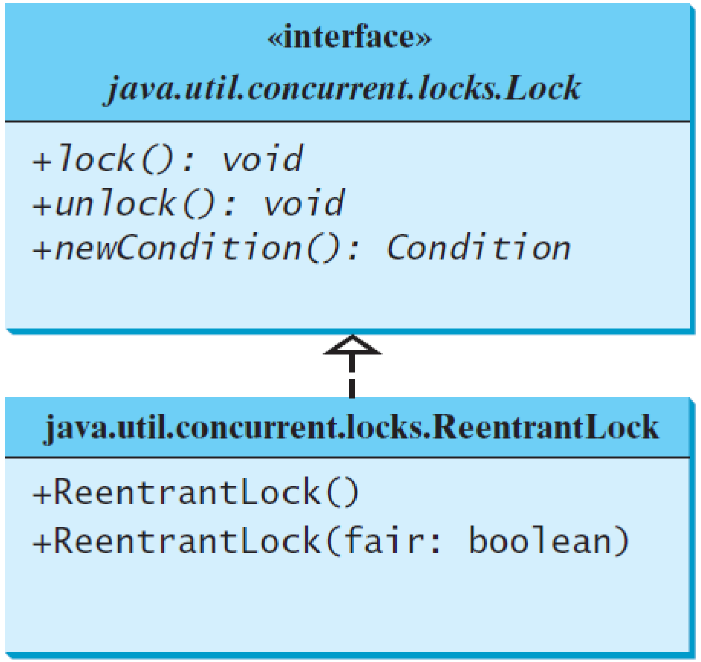
c_1: buffer is non-empty

- consumer tasks queue for c_1

c_2: buffer is not full

- supplier tasks queue for c_2

In contrast: implicit lock – one lock and one condition per object (more restrictive)



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

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

java.util.concurrent.locks.ReentrantLock

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

Need to call these methods explicitly

- Besides using monitor-based locks, we can synchronize threads by explicit locks
- An explicit lock is an instance of the Lock interface, which declares the methods to acquire and release locks. A lock may use the newCondition() method to create any number of Condition objects, which can be used for thread communications

Fairness Policy

- ReentrantLock is a concrete implementation of Lock for creating mutual exclusive locks.
  - There is no effect to a thread when it acquires a reentrant lock being held
- We can create a lock with the specified fairness policy.
  - True fairness guarantees the longest-wait thread to obtain the lock first
  - False fairness grants a lock to a waiting thread without any access order
  - Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee free from starvation

Using Explicit Locks

```
public class Account {
  private Lock lock = new ReentrantLock(true); // Create a fair lock
  …
  public void deposit(int amount) throws InterruptedException {
    lock.lock(); // Acquire the lock
    try {
      var newBalance = balance + amount;
      Thread.sleep( 5 );
      balance = newBalance;
    } catch (InterruptedException ex) {
        throws ex;
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
}
```

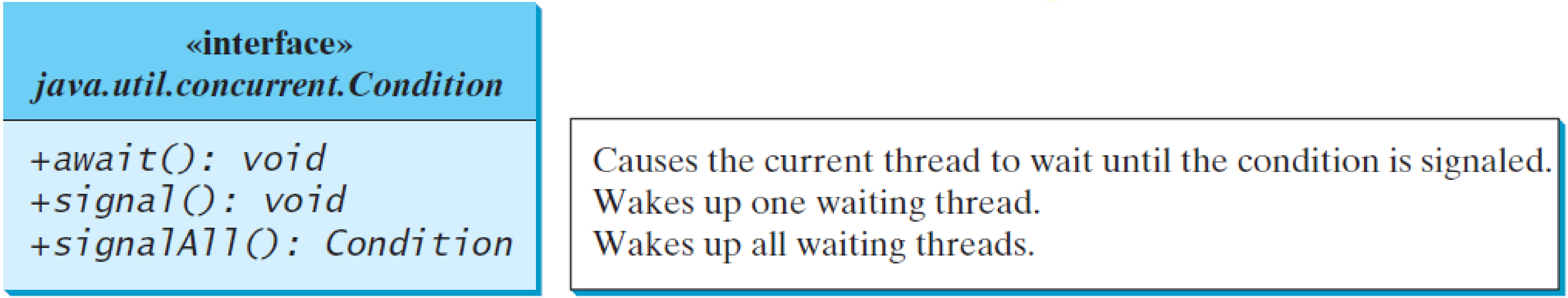
critical region

Reentrant lock:
  lock()
  unlock()

AccountWithSyncUsingLock.java

Why do we need to call unlock in finally block?

Cooperation Among Threads



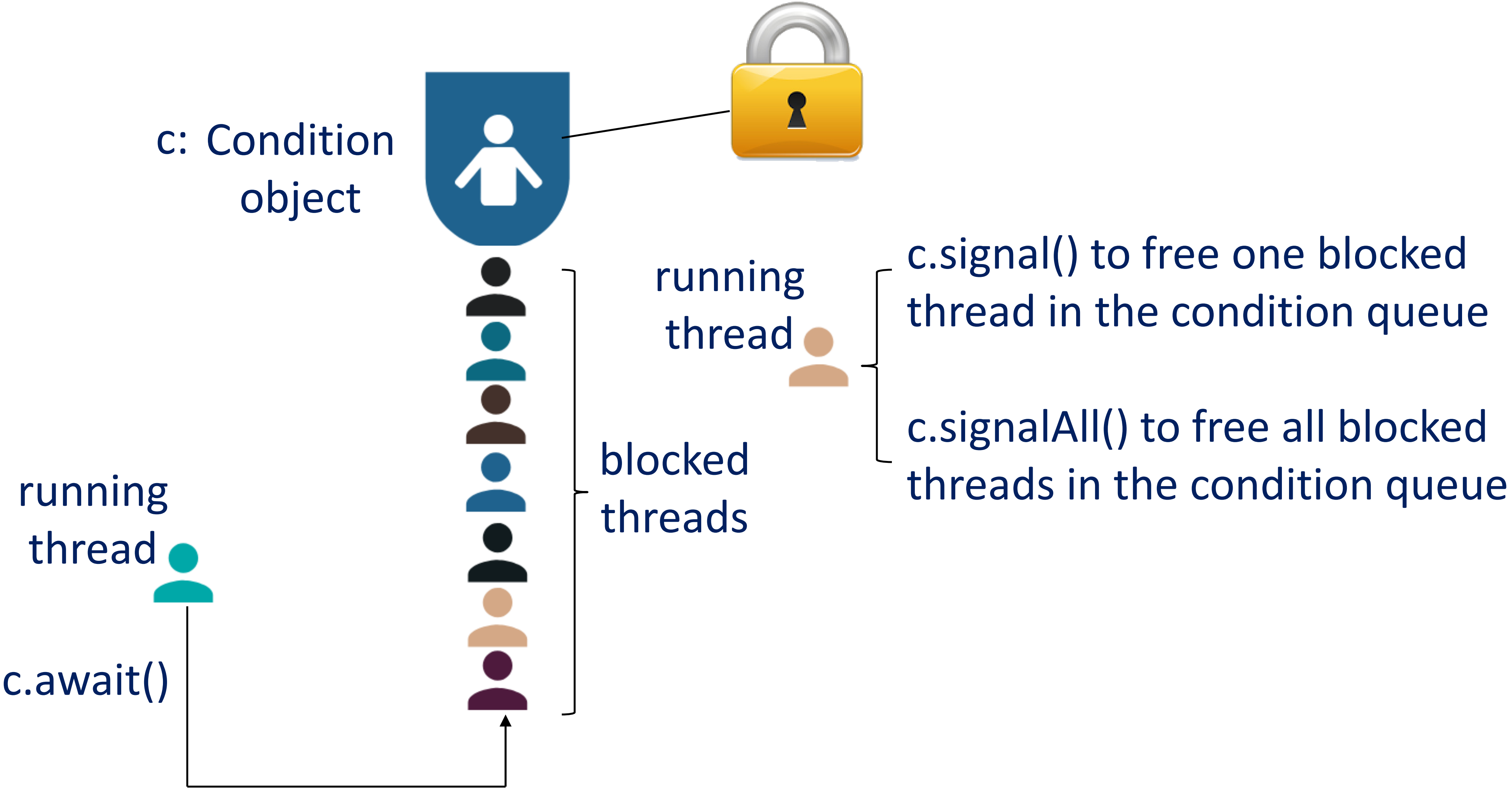
«interface»
java.util.concurrent.Condition

```
+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.

- The conditions facilitate communications among threads. A thread can specify what to do under each condition
- Conditions are objects created by invoking the newCondition() method on a Lock object. Once a condition is created, we use its await(), signal(), and signalAll() methods for thread communications
- await() causes the current thread to wait until the condition is signaled; signal() wakes up one waiting thread, and signalAll() wakes up all waiting threads

Condition – await(), signal(), signalAll()


c: Condition object
running thread
c.await()
blocked threads
running thread
c.signal() to free one blocked thread in the condition queue
c.signalAll() to free all blocked threads in the condition queue

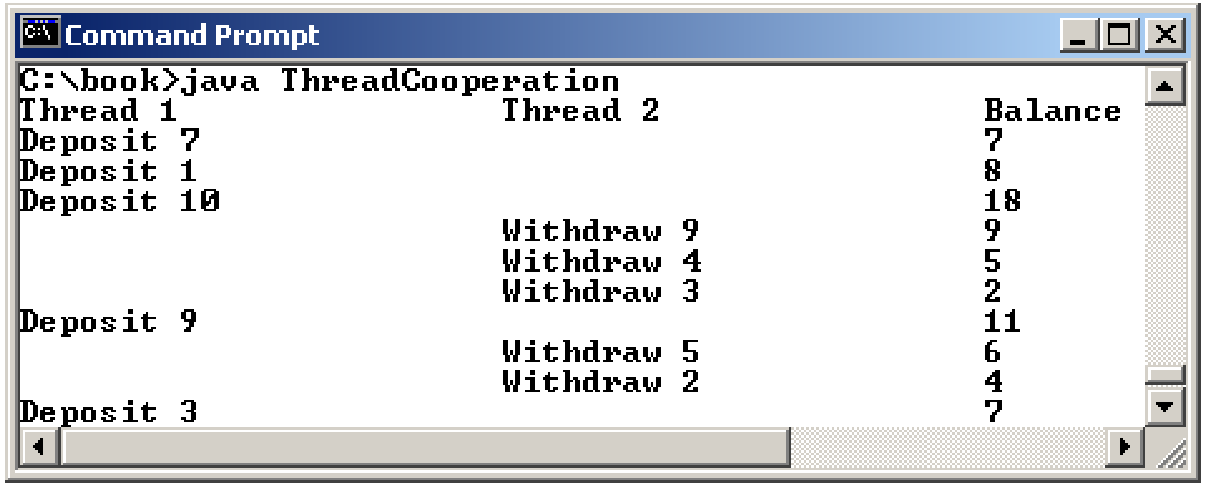
Example: Thread Cooperation

Suppose we launch two threads: one deposits to an account, and the other withdraws from the same account.

The withdraw thread awaits if the amount to be withdrawn is over the current account balance.

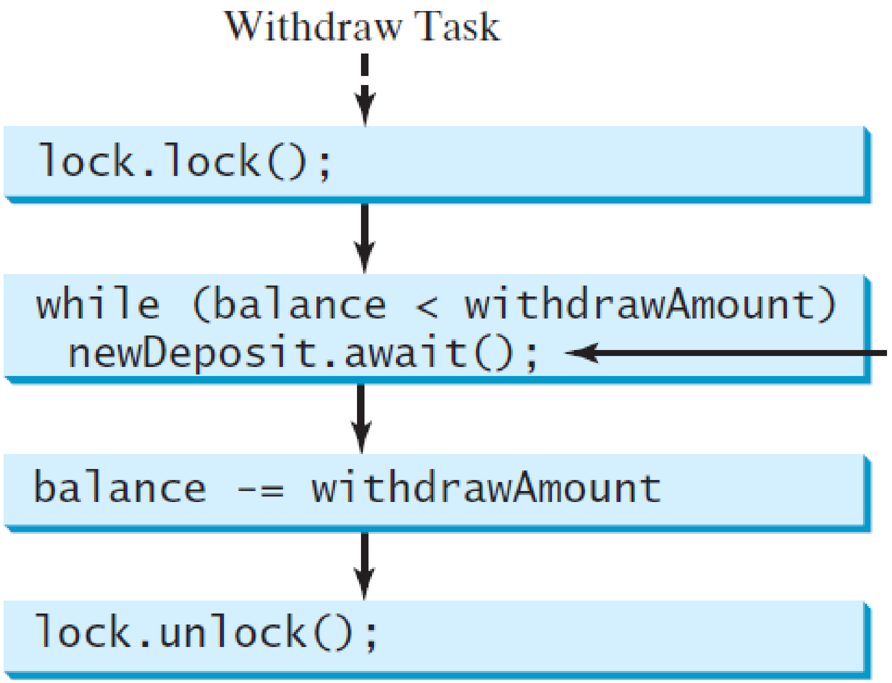
Whenever the deposit thread adds new fund to the account, it signals the withdraw thread to resume. If the amount is still not enough for the withdrawal, the withdraw thread continues to await more fund in the account.

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                                       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
```

Cooperation Among Threads



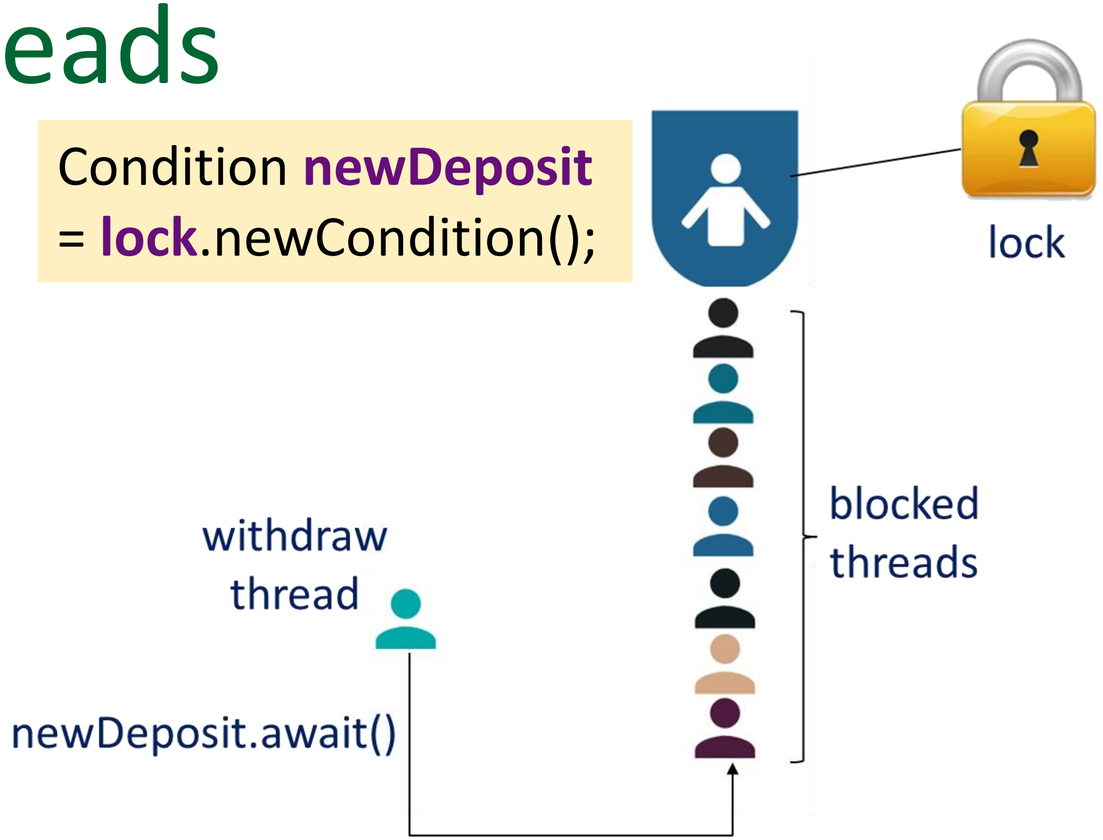
Withdraw Task

```
lock.lock();

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

balance -= withdrawAmount

lock.unlock();
```



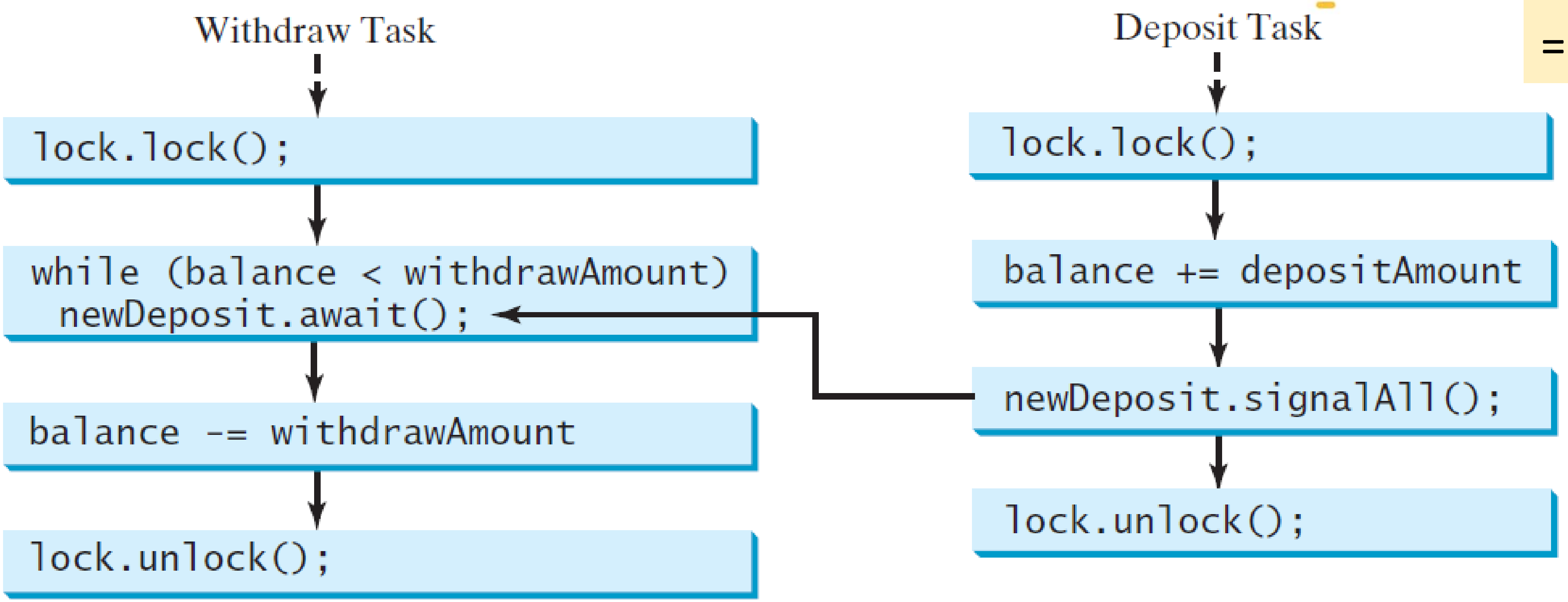
Condition **newDeposit** = **lock**.newCondition();

- Use a lock with a condition: newDeposit (to indicate 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

ThreadCooperation.java

Cooperation Among Threads

Condition **newDeposit** = **lock**.newCondition();



Withdraw Task

```
lock.lock();
while (balance < withdrawAmount)
  newDeposit.await();
balance -= withdrawAmount
lock.unlock();
```

Deposit Task

```
lock.lock();
balance += depositAmount
newDeposit.signalAll();
lock.unlock();
```

- Use a lock with a condition: newDeposit (to indicate 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

ThreadCooperation.java

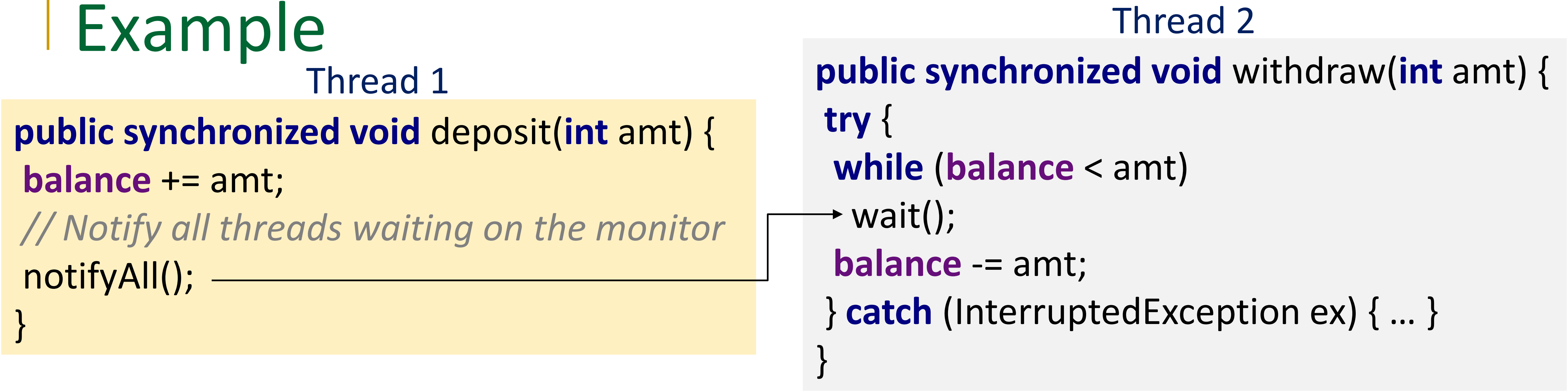
Java’s Built-in Monitors (Implicit Locks)

- Prior to Java 5, thread communications are programmed using object’s built-in monitors.
- Locks and conditions introduced since Java 5 are more powerful and flexible than the built-in monitor, which assumes one condition for each monitor lock
- However, we likely encounter the Java’s built-in monitor when working with legacy code.
- 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 to execute a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor

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

- Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.
- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur
- The wait() method lets the thread wait until some condition occurs and immediately release the monitor lock of the object that owns the wait() method
- When the condition occurs, we 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
- When a waiting thread wakes up, it re-acquires the monitor lock that it has released

Example



Thread 1

```
public synchronized void deposit(int amt) {
 balance += amt;
 // Notify all threads waiting on the monitor
 notifyAll();
}
```

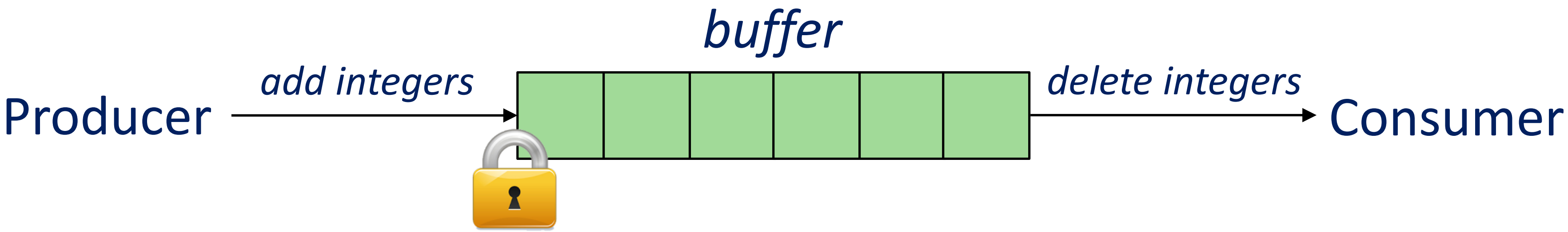
Thread 2

```
public synchronized void withdraw(int amt) {
 try {
  while (balance < amt)
   wait();
  balance -= amt;
 } catch (InterruptedException ex) { ... }
}
```

- When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread wakes up after being notified, the lock is automatically re-acquired
- The wait(), notify(), and notifyAll() methods on a monitor object are analogous to the await(), signal(), and signalAll() methods on a condition.

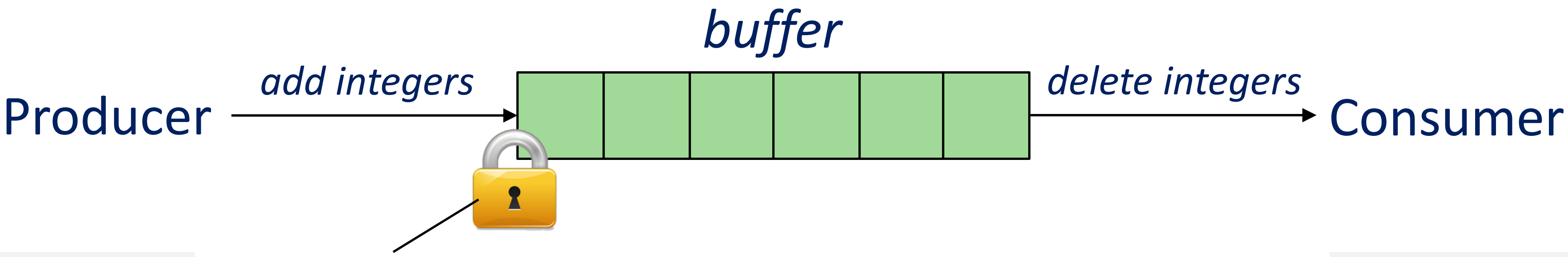
ThreadCooperationUsingBuiltInMonitor.java

Why replacing implicit locking by explicit locking?



- Implicit locking assumes one condition for each lock
  - Each implicit lock supports one thread waiting queue (i.e., one condition)
- Explicit locking allows multiple conditions created for one lock
  - Each condition supports one thread waiting queue
- There are many applications where accesses to a critical section are subject to multiple conditions. Example: Producer/Consumer

Producer/Consumer: collaborate on a lock with two conditions



Condition 1:

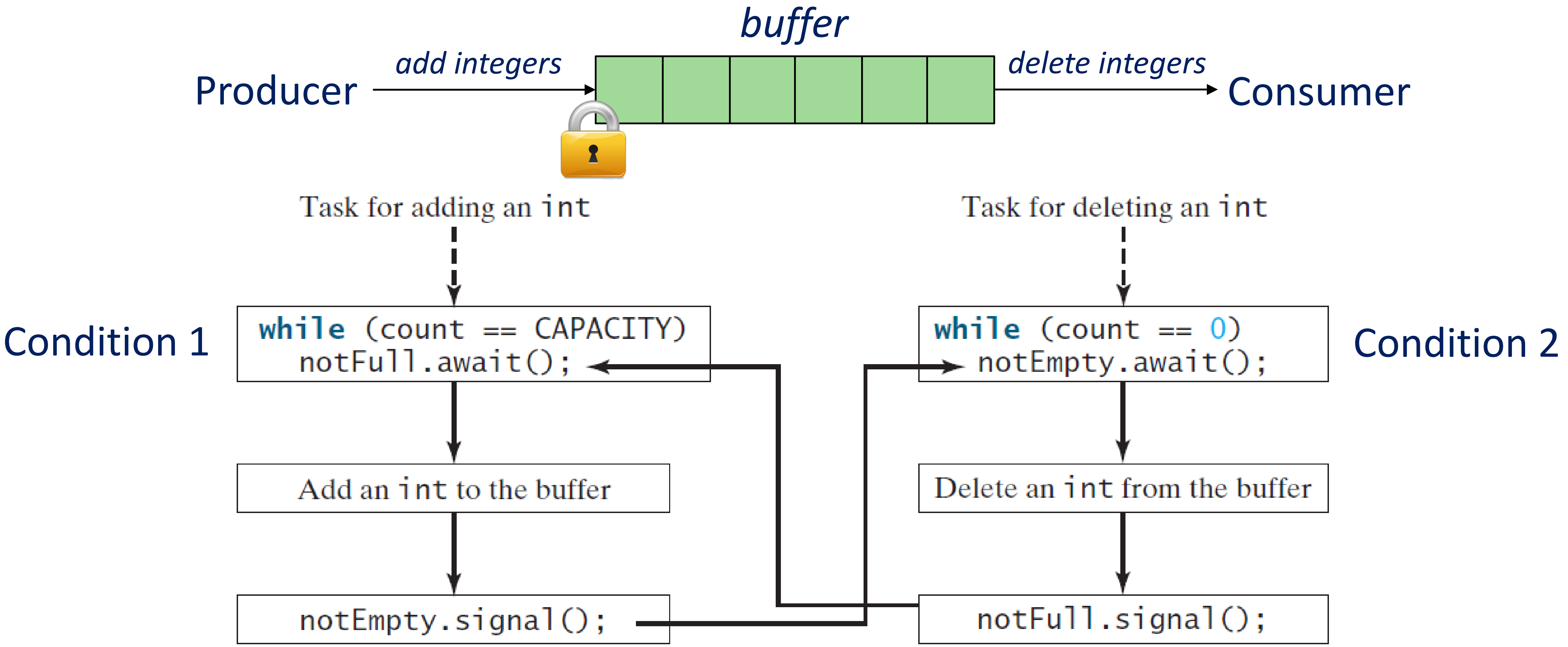
- Producer may add integers only if the buffer is not full
- Producers queue at this condition when the buffer is full

- One lock to protect buffer updates: *Only one thread can update the buffer at any time*
- Buffer updates are governed by two conditions

Condition 2:

- Consumer may delete integers only if the buffer is not empty
- Consumers queue at this condition when the buffer is empty

Producer/Consumer: collaborate on a lock with two conditions



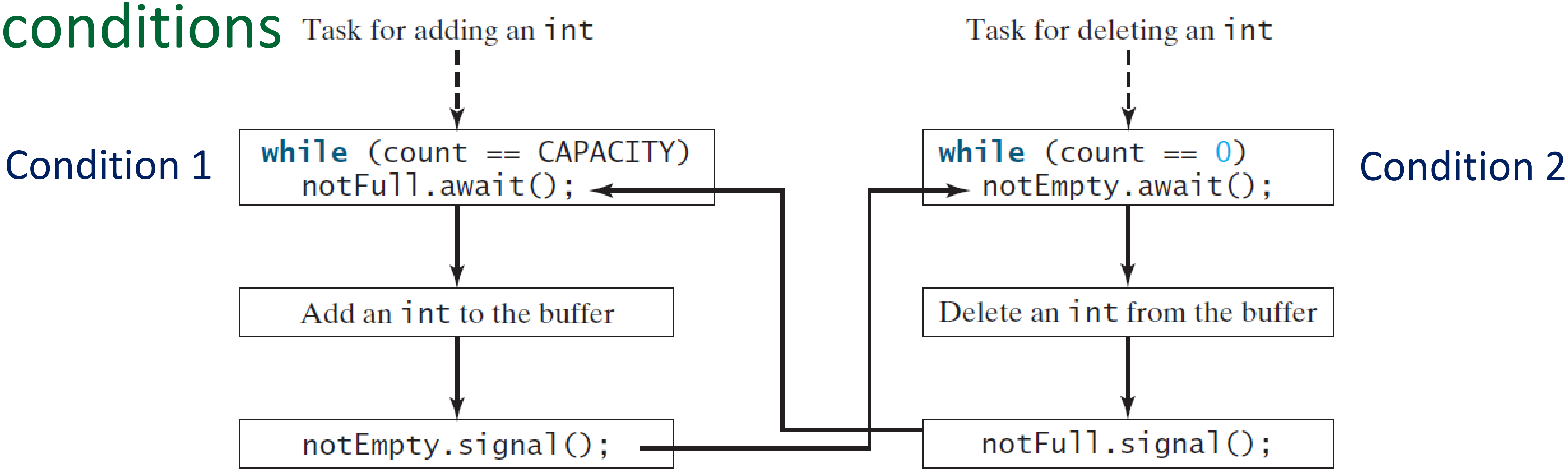
```
while (count == CAPACITY)
  notFull.await();
```

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

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

```
notFull.signal();
```

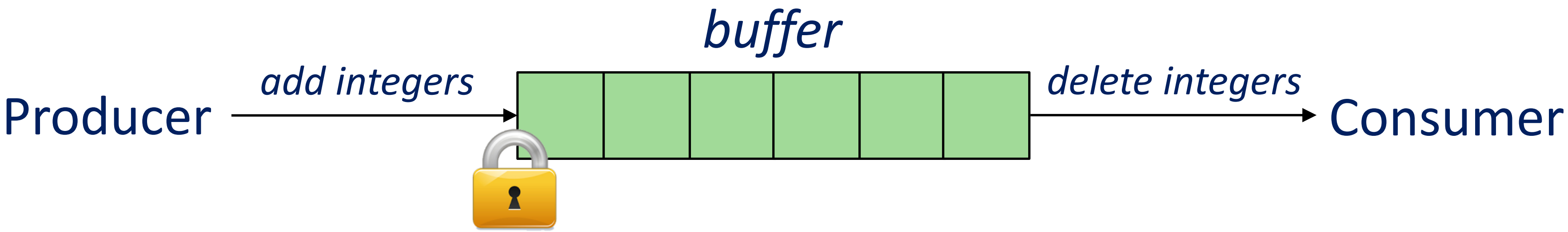
Producer/Consumer: collaborate on a lock with two conditions



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.

ConsumerProducerUsingLock.java

Final Note



- It is sometimes possible to emulate one explicit locks with *N* conditions by *N* monitor-based locks
- The solution is usually less elegant and less maintainable
- This problem is more obvious in solutions that require more than one explicit lock
- Advice: Use explicit locking but able to understand monitor-based locking