CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Java
Multithreading and Concurrency

Thread

- Thread: single sequential flow of control within a program
- Single-threaded program can handle one task at any time.
- Multitasking allows single processor to run several concurrent threads.
- Most modern operating systems support multitasking.
- A thread can execute concurrently with other threads within a single process.
- All threads managed by the JVM share memory space and can communicate with each other.

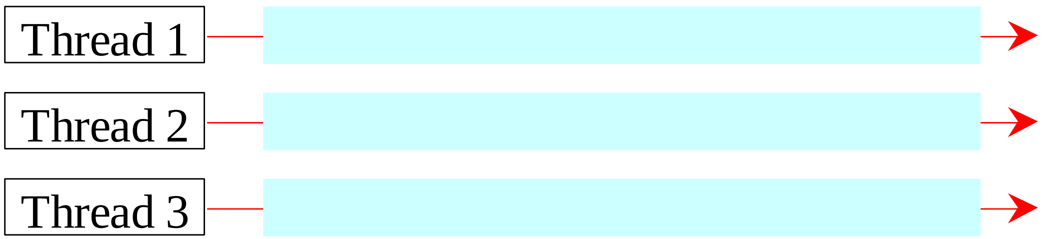
Advantages of Multithreading

- Reactive systems – constantly monitoring
- More responsive to user input – GUI application can have a thread continuously interacting with the user
- Server can handle multiple clients simultaneously
- Can take advantage of concurrency

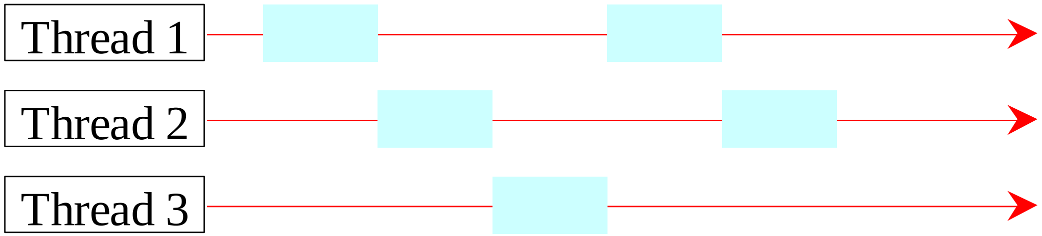
- What about on a single core?

Threads Concept

Multiple threads on multiple cores



Multiple threads sharing a single core



Threads in Java

Creating threads in Java:

- Extend java.lang.Thread class

OR

- Implement java.lang.Runnable interface

Threads in Java

Creating threads in Java:

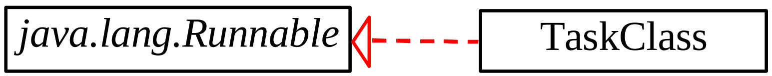
- Extend java.lang.Thread class
  - run() method must be overridden (similar to main method of sequential program)
  - run() is called when execution of the thread begins
  - A thread terminates when run() returns
  - start() method invokes run()
  - Calling run() does not create a new thread
- Implement java.lang.Runnable interface

Threads in Java

Creating threads in Java:

- Extend java.lang.Thread class
- Implement java.lang.Runnable interface
  - If already inheriting another class
  - Single method: public void run()
  - Thread class implements Runnable.

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
    ...
  }
  ...
}
```

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

The Thread Class

«interface»
java.lang.Runnable

java.lang.Thread

java.lang.Thread	
+Thread()	Creates a default 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.
+<u>sleep(millis: long): void</u>	Puts the runnable object to sleep for a specified time in milliseconds.
+<u>yield(): void</u>	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

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

- TaskThreadDemo.java: Create and run 3 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 to 100.

The Static yield() Method

You can use the yield() method to temporarily release time for other threads. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:

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

Every time a number is printed, the print100 thread is yielded. So, the numbers are printed after the characters.

Is this correct?

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:

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

Every time a number (>= 50) is printed, the <u>print 100</u> thread is put to sleep for 1 millisecond.

What is the effect?

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:

```
blic void run() {
Thread printC = new Thread(
  new PrintChar('c', 40));
printC.start();
try {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    if (i == 50) printC.join();
  }
}
catch (InterruptedException ex) {
}
```

Thread print100

Thread printC

printC.join()

Wait for printC to finish

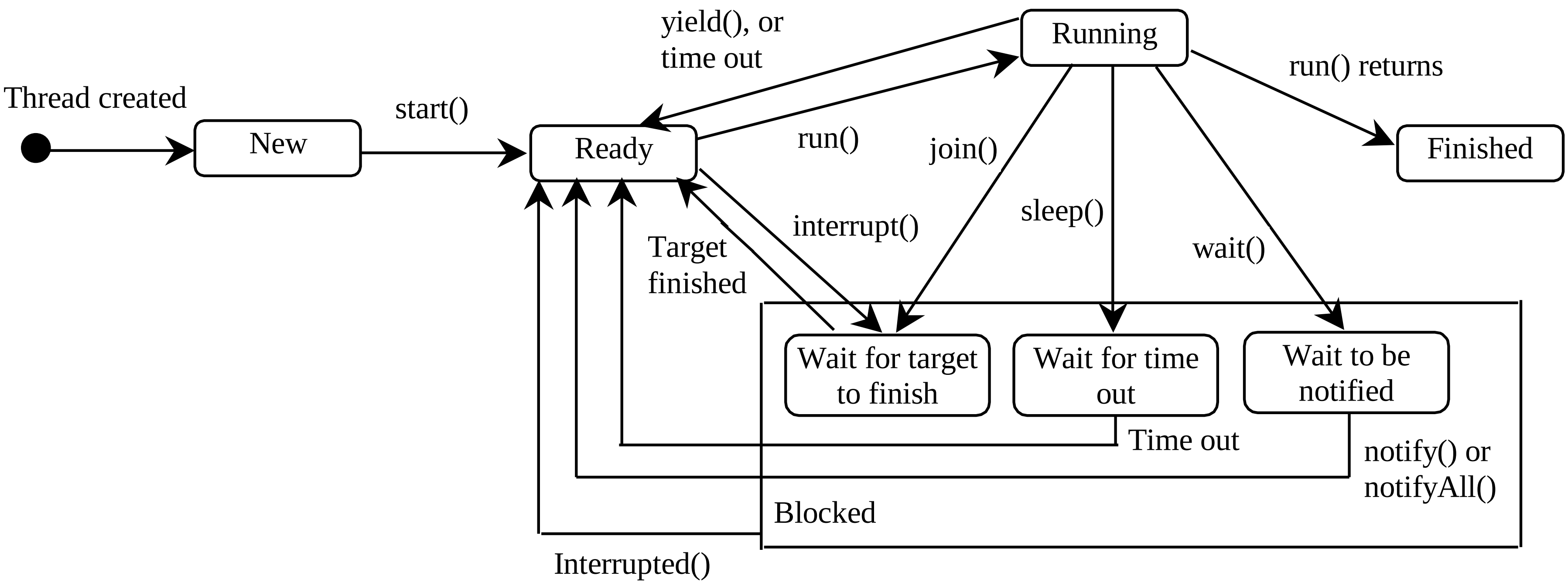
printC finished

The numbers after 50 are printed after thread printC is finished.

Is this correct?

Thread States

A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.



Thread methods

isAlive()

- method used to find out the state of a thread.
- returns true: thread is in the Ready, Blocked, or Running state
- returns false: thread is new and has not started or if it is finished.

interrupt()

- if a thread is currently in the Ready or Running state, its interrupted flag is set;
- if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.
- isInterrupt() tests whether the thread was interrupted.

Thread Priority

- Each thread is assigned a default priority of `Thread.NORM_PRIORITY` (constant of 5). You can reset the priority using `setPriority(int priority)`.

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

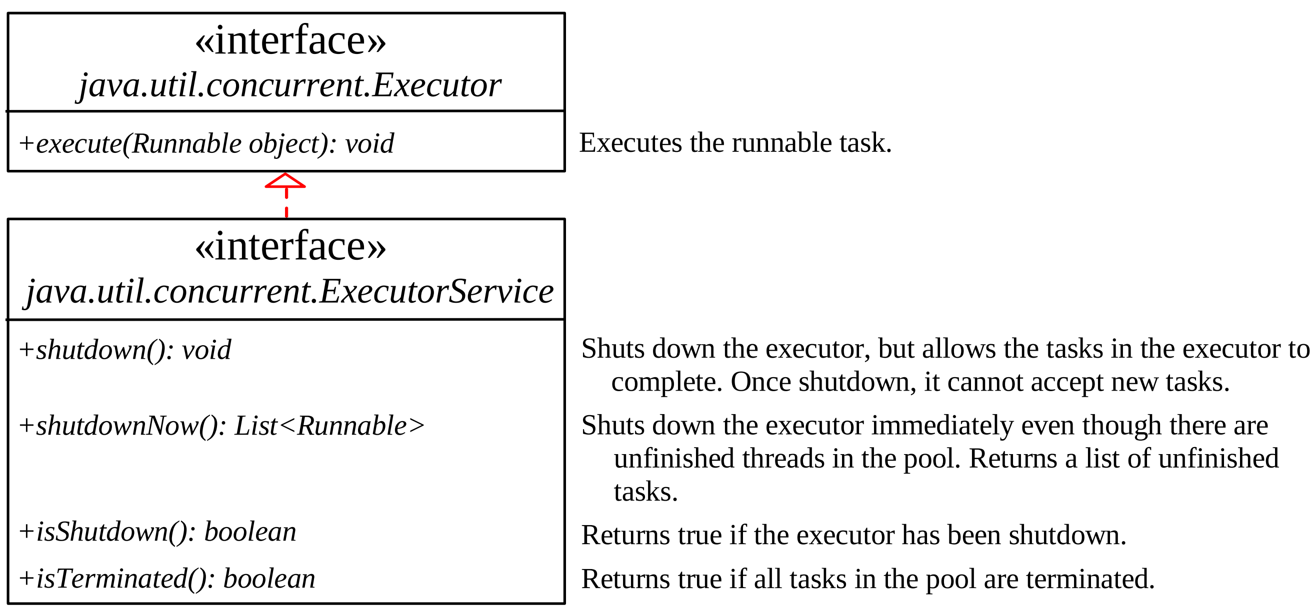
- By default, a thread has the priority level of the thread that created it.

Thread Scheduling

- An operating system's thread scheduler determines which thread runs next.
- Most operating systems use *timeslicing* for threads of equal priority.
- *Preemptive scheduling*: when a thread of higher priority enters the running state, it preempts the current thread.
- *Starvation*: Higher-priority threads can postpone (possible forever) the execution of lower-priority threads.

Thread Pools

- Starting a new thread for each task could limit throughput and cause poor performance.
- A thread pool is ideal to manage the number of tasks executing concurrently.
- Executor interface for executing Runnable objects in a thread pool
- ExecutorService is a subinterface of Executor.



Creating Executors

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

java.util.concurrent.Executors	
+newFixedThreadPool(numberOfThreads: int): ExecutorService	Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.
+newCachedThreadPool(): ExecutorService	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

Executor Demo

```
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();
  }
}
```

Question?

- Why don't we call run() method directly, why call start() method?
  - We can call run() method if we want but then it would behave just like a normal method and we would not be able to take the advantage of **multithreading**.
  - When the run method gets called though start() method then a new separate thread is being allocated to the execution of run method, so if more than one thread calls start() method that means their run method is being executed by separate threads (these threads run simultaneously).

Thread Synchronization

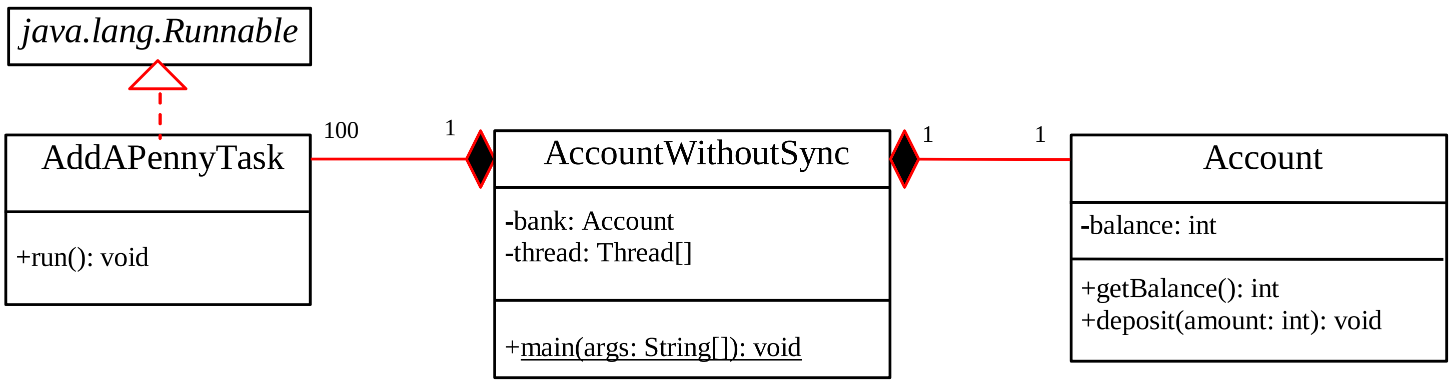
A shared resource may be corrupted if it is accessed simultaneously by multiple threads.

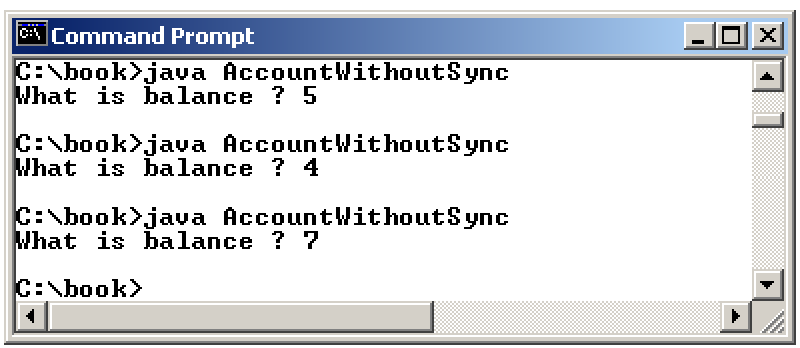
Example: two unsynchronized threads accessing the same bank account may cause conflict.

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

- AccountWithoutSync: 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.





Race Condition

What, then, caused the error in the example? Here is a possible scenario:

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

- Effect: Task 1 did nothing (in Step 4 Task 2 overrides the result)
- Problem: Task 1 and Task 2 are accessing a common resource in a way that causes conflict.
- Known as a *race condition* in multithreaded programs.
- A *thread-safe* class does not cause a race condition in the presence of multiple threads.
- The Account class is not thread-safe.

Synchronization Using Locks

- You can use locks explicitly to obtain more control for coordinating threads.
- A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks.
- newCondition() method creates Condition objects, which can be used for thread communication.

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

+lock(): void — Acquires the lock.
+unlock(): void — Releases the lock.
+newCondition(): Condition — Returns a new Condition instance that is bound to this Lock instance.

java.util.concurrent.locks.ReentrantLock

+ReentrantLock() — Same as ReentrantLock(false).
+ReentrantLock(fair: boolean) — 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.

Fairness Policy

- ReentrantLock: Concrete implementation of Lock for creating mutually exclusive locks.
- Create a lock with the specified fairness policy.
  - True fairness policies guarantee the longest-wait thread to obtain the lock first.
  - False fairness policies grant a lock to a waiting thread without any access order.

- Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Example: Using Locks

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

Cooperation Among Threads

- Conditions can be used for communication among threads.
- A thread can specify what to do under a certain condition.
- newCondition() method of Lock object.

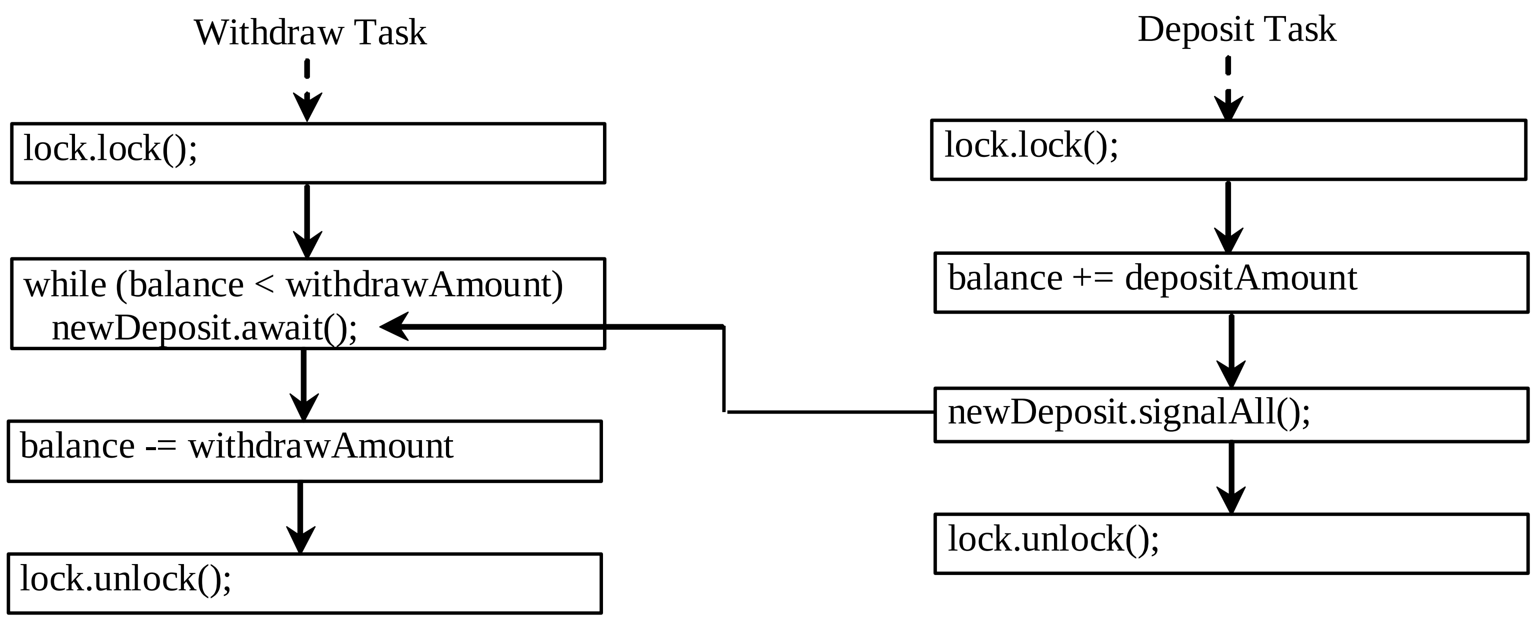
- Condition methods:
  - await() current thread waits until the condition is signaled
  - signal() wakes up a waiting thread
  - signalAll() wakes all waiting threads

«interface» *java.util.concurrent.Condition*	
+await(): void	Causes the current thread to wait until the condition is signaled.
+signal(): void	Wakes up one waiting thread.
+signalAll(): Condition	Wakes up all waiting threads.

Cooperation Among Threads

- Lock with a condition to synchronize operations: newDeposit
- 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.

- Interaction between the two tasks:



Example: Thread Cooperation

ThreadCooperation: It demonstrates thread cooperation. Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund is deposited to the account, the first thread notifies the second thread to resume. If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account. Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.

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

Java’s Built-in Monitors

- Locks and conditions are introduced in Java 5.
- Prior to Java 5, thread communications were programmed using object’s built-in monitors.
- Locks and conditions are more powerful and flexible than the built-in monitor.
- A *monitor* is an object with mutual exclusion and synchronization capabilities with implicit locking mechanism.
- Only one thread can execute a method at a time in the monitor.
- A thread enters the monitor by acquiring a lock (<u>synchronized</u> keyword on method / block) on the monitor and exits by releasing the lock.
- A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.
- *Any object can be a monitor*. An object becomes a monitor once a thread locks it.

`synchronized`

- Synchronization keeps other threads waiting until the object is available.
- The synchronized keyword synchronizes the method so that only one thread can access the method at a time.
- For example, one way to correct the problem in example, make Account thread-safe by adding the synchronized keyword in deposit:

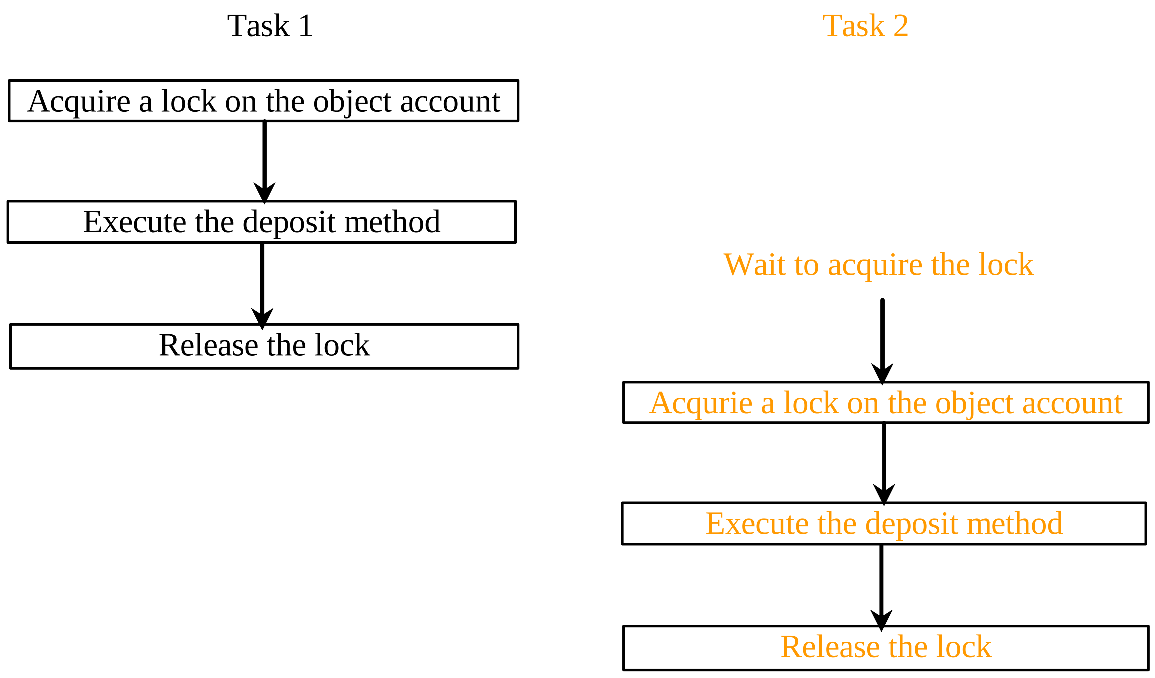
```
public synchronized void deposit(double amount)
```

Synchronizing Methods

- A synchronized method acquires a lock before it executes.
  - Instance method: the lock is on the object for which it was invoked.
  - 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, 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 Methods

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



Synchronizing Statements

- Invoking a synchronized instance method of an object acquires a lock on the object.
- Invoking a synchronized static method of a class acquires a lock on the class.
- A *synchronized block* can be used to acquire a lock on any object, not just *this* object, when executing a block of code.

```
synchronized (expr) {
  statements;
}
```

- expr must evaluate to an object reference.
- If the object is already locked by another thread, the thread is blocked until the lock is released.
- When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronizing Statements vs. Methods

Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

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

This method is equivalent to

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

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

Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.

The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

Example: Using Monitor

Task 1

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

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

resume

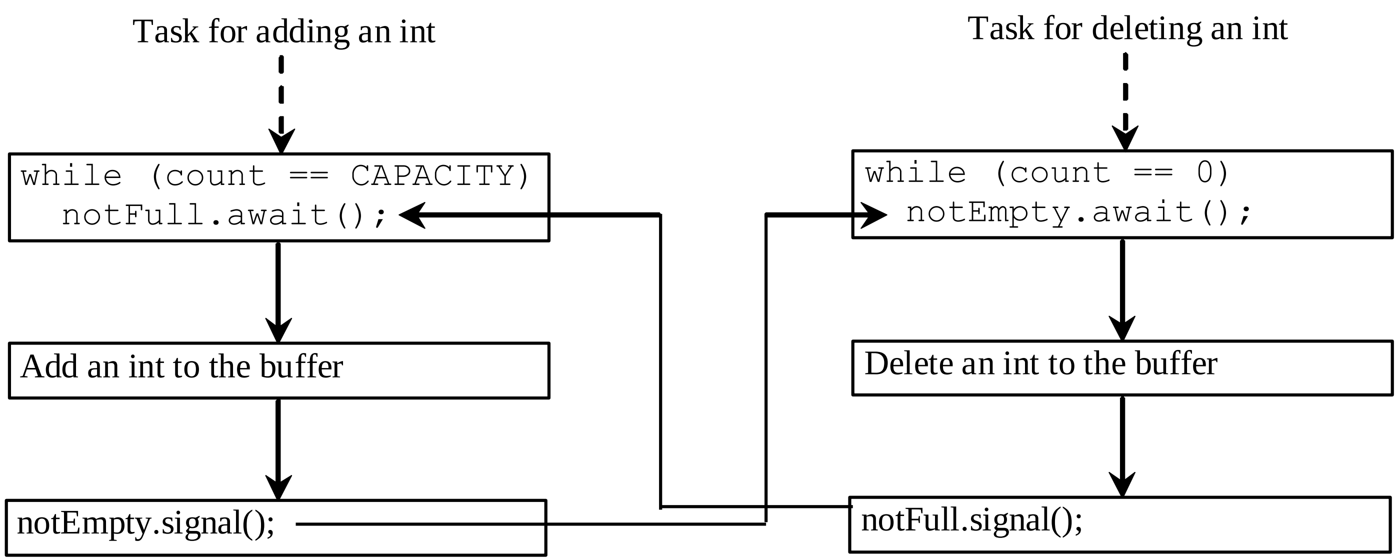
Task 2

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

- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an IllegalMonitorStateException will occur.
- When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.
- The wait(), notify(), and notifyAll() methods on an object are analogous to the await(), signal(), and signalAll() methods on a condition.

Case Study: Producer/Consumer

Consider the classic Consumer/Producer example. Suppose you use a buffer to store integers. The buffer size is limited. The buffer provides the method write(int) to add an int value to the buffer and the method read() to read and delete an int value from the buffer. To synchronize the operations, use a lock with two conditions: notEmpty (i.e., buffer is not empty) and notFull (i.e., buffer is not full). When a task adds an int to the buffer, if the buffer is full, the task will wait for the notFull condition. When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the notEmpty condition.



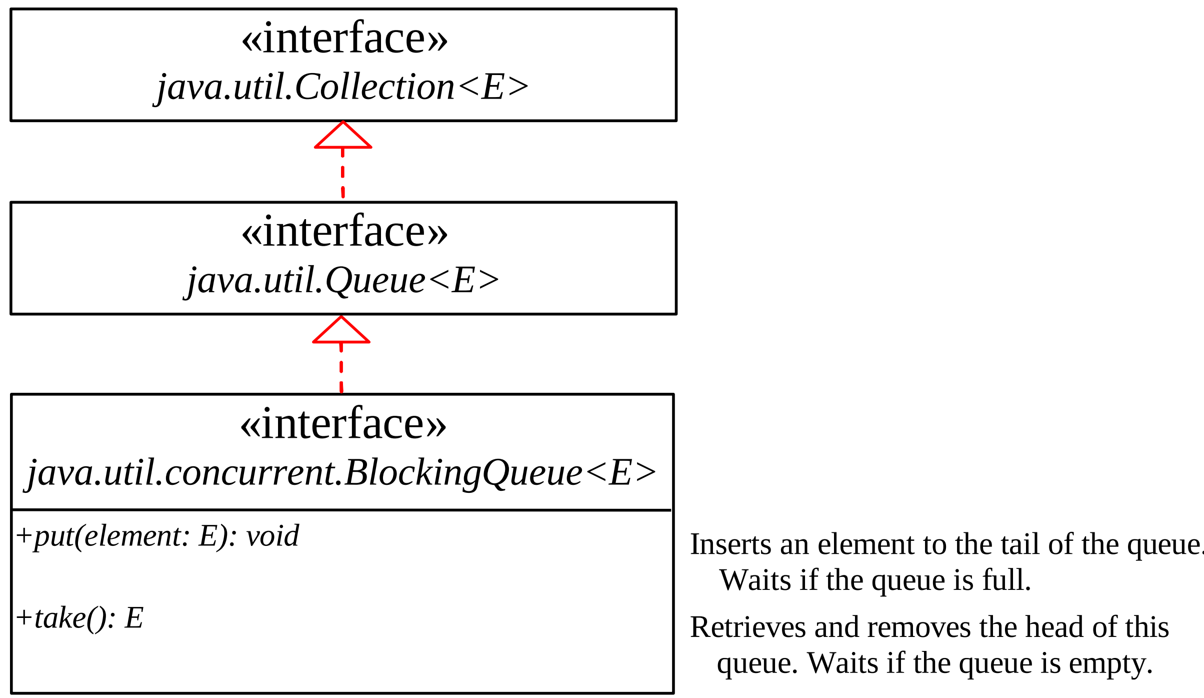
Case Study: Producer/Consumer

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

For simplicity, the buffer is implemented using a linked list (lines 48-49). Two conditions notEmpty and notFull on the lock are created in lines 55-56. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the wait() and notify() methods to rewrite this example, you have to designate two objects as monitors.

Blocking Queues

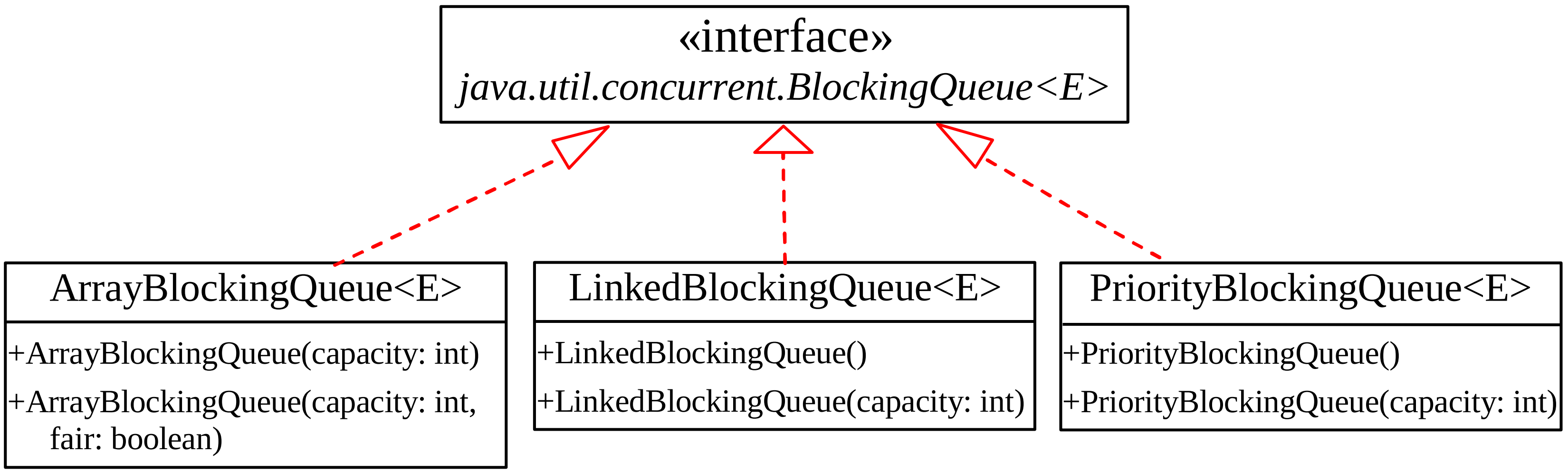
A *blocking queue* causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue.



Concrete Blocking Queues

Three concrete blocking queues are in the java.util.concurrent package.

- ArrayBlockingQueue implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an ArrayBlockingQueue.
- LinkedBlockingQueue implements a blocking queue using a linked list. You may create an unbounded or bounded LinkedBlockingQueue.
- PriorityBlockingQueue is a priority queue. You may create an unboundedor bounded priority queue.

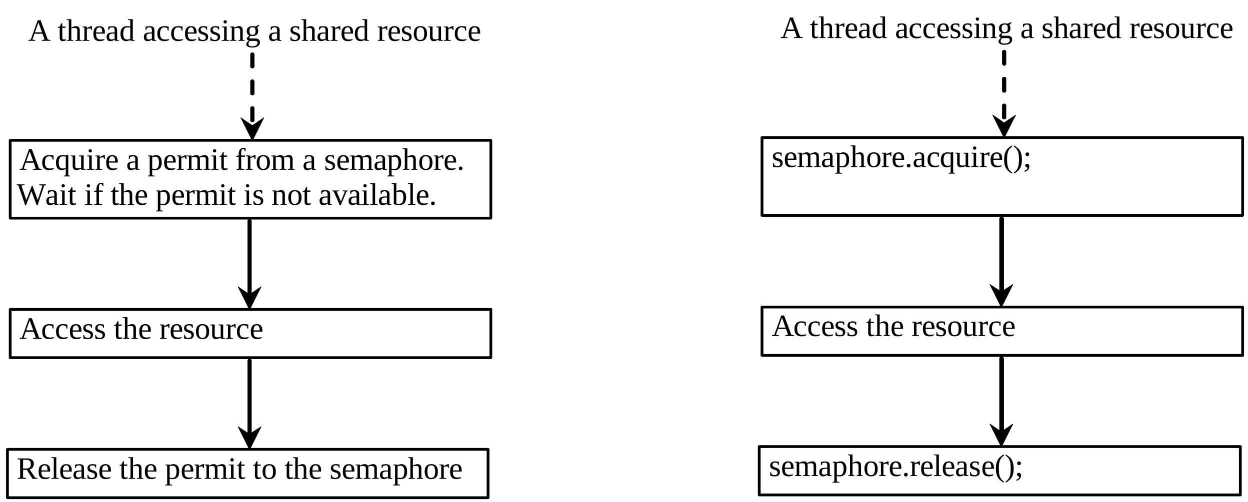


Producer/Consumer Using Blocking Queues

ConsumerProducerUsingBlockingQueue.java: It gives an example of using an ArrayBlockingQueue to simplify the Consumer/Producer example.

Semaphores

Semaphores can be used to restrict the number of threads that access a shared resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore.



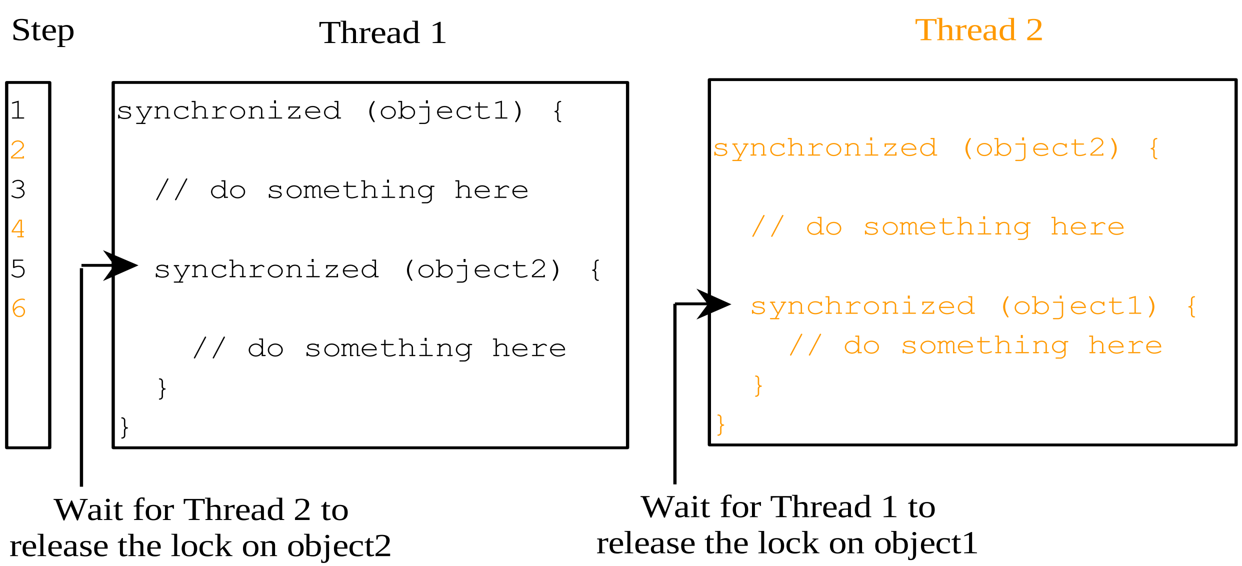
Creating Semaphores

To create a semaphore, you have to specify the number of permits with an optional fairness policy. A task acquires a permit by invoking the semaphore's acquire() method and releases the permit by invoking the semaphore's release() method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1. Once a permit is released, the total number of available permits in a semaphore is increased by 1.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

Deadlock

- Sometimes two or more threads need to acquire the locks on several shared objects.
- This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object.
- In the figure below, the two threads wait for each other to release the in order to get a lock, and neither can continue to run.



Preventing Deadlock

- Deadlock can be easily avoided by resource ordering.
- With this technique, assign an order on all the objects whose locks must be acquired and ensure that the locks are acquired in that order.

Synchronized Collections

- The classes in the Java Collections Framework are not thread-safe.
- Their 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 <u>Collections</u> class provides six static methods for creating *synchronization wrappers*.

java.util.Collections	
+<u>synchronizedCollection(c: Collection): Collection</u>	Returns a synchronized collection.
+<u>synchronizedList(list: List): List</u>	Returns a synchronized list from the specified list.
+<u>synchronizedMap(m: Map): Map</u>	Returns a synchronized map from the specified map.
+<u>synchronizedSet(s: Set): Set</u>	Returns a synchronized set from the specified set.
+<u>synchronizedSortedMap(s: SortedMap): SortedMap</u>	Returns a synchronized sorted map from the specified sorted map.
+<u>synchronizedSortedSet(s: SortedSet): SortedSet</u>	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:

```
public boolean add(E o) {
  synchronized (this) { return c.add(o); }
}
```

The synchronized collections can be safely accessed and modified by multiple threads concurrently.

Note: The methods in java.util.Vector, java.util.Stack, and Hashtable are already synchronized, but not much recommended as they are old classes of 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 them with 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:

```
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.