Code.Hub

The first Hub for Developers

Multithreading, Concurrency in Java

Threads

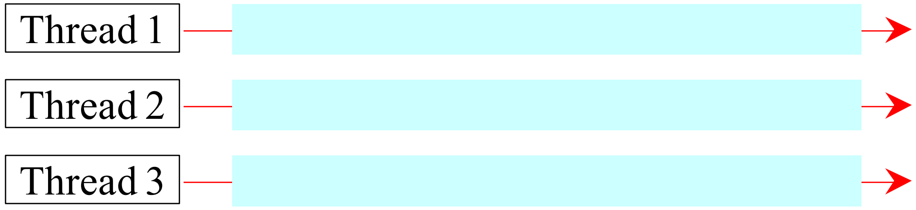
- Threads are lightweight processes as the overhead of switching between threads is less
- The can be easily spawned
- The Java Virtual Machine spawns a thread when your program is run called the Main Thread

Why do we need threads?

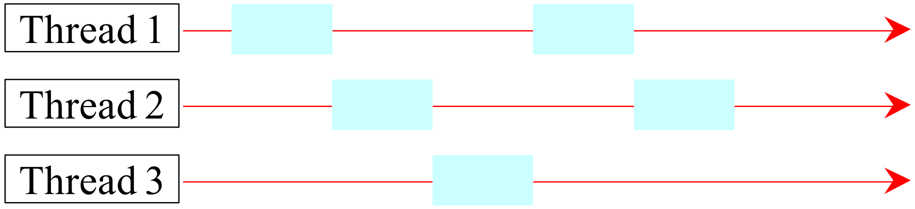
- To increase response to the user
- To utilize the idle time of the CPU
- Prioritize your work depending on priority

Threads Concept

Multiple threads on multiple CPUs



Multiple threads sharing a single CPU



Implementing threads

Example

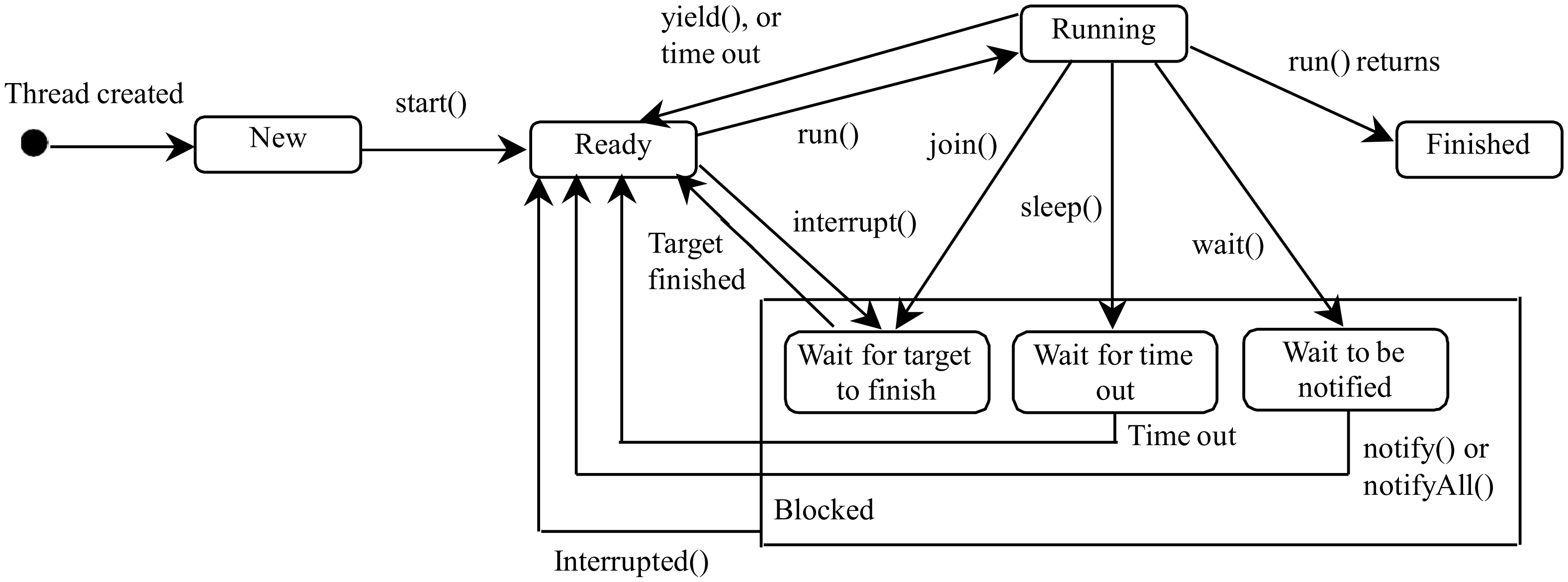
```
public class myThread implements Runnable{
        private static long sleepTime = 100;
        private String name;
        public myThread (String name) { this.name=name;}
        public void run() {
                for (int i=0;i<100;i++) {
                    System.out.println(“Thread “+name +” iterates at ”+i);
                    try {   Thread.sleep(sleepTime); } catch (InterruptedException e) {}
                 }  }   }

public class mainClass {
        public static void main(String args[]) {
                     Thread t1 = new Thread(new myThread( “Thread No. 1”));
                     Thread t2 = new Thread(new myThread( “Thread No. 2”));
                     t1.run();  t2.run();
                     t1.start();  t2.start();
        } }
```

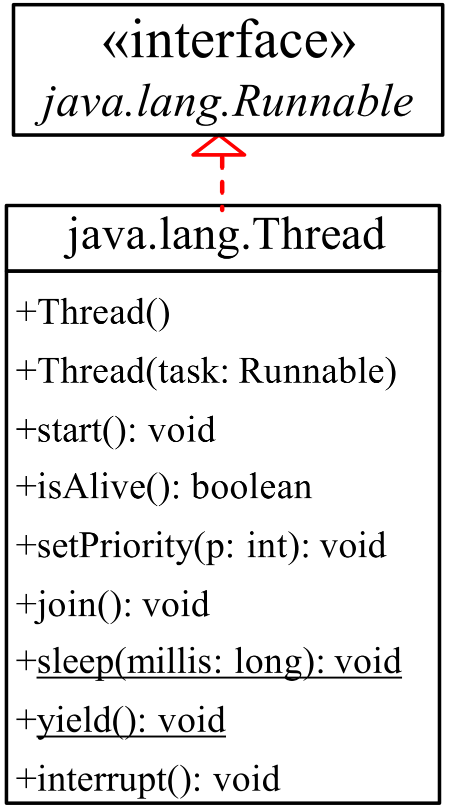
- Calling t.run() does not start a thread, it is just a simple method call.
- Creating an object does not create a thread, calling start() method creates the thread.

Thread States

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



The Thread Class



«interface»
java.lang.Runnable

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.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

The Static yield() Method

You can use the yield() method to temporarily release time for other threads. For example, suppose you modify the code 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 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,

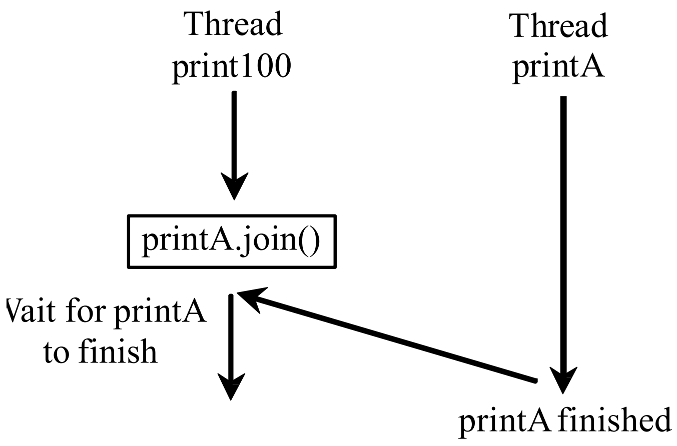
```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

Every time a number (>= 50) is printed, the 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

```
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 printA is 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()

f a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.

The isInterrupt() method tests whether the thread is interrupted.

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.

Question

Why do you think that these methods were deprecated?

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`

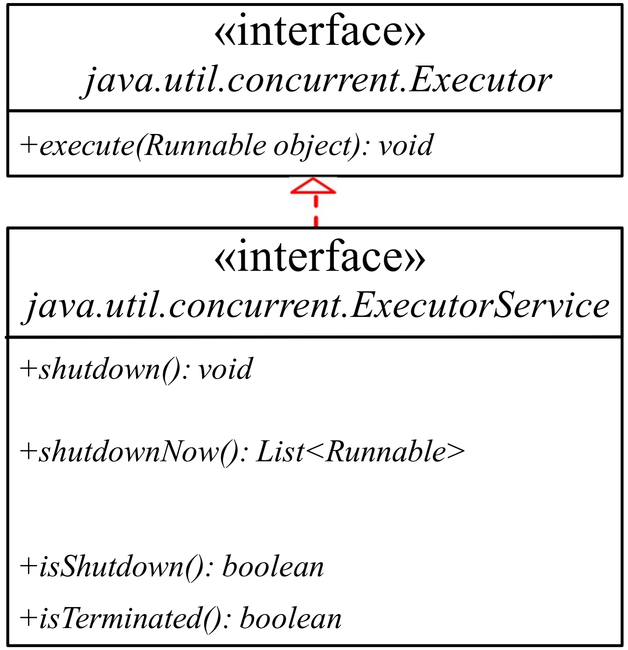
- By default, a thread has the priority level of the thread that created it.

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.


«interface»
java.util.concurrent.Executor
+execute(Runnable object): void
«interface»
java.util.concurrent.ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+isTerminated(): boolean

Executes the runnable task.

Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.

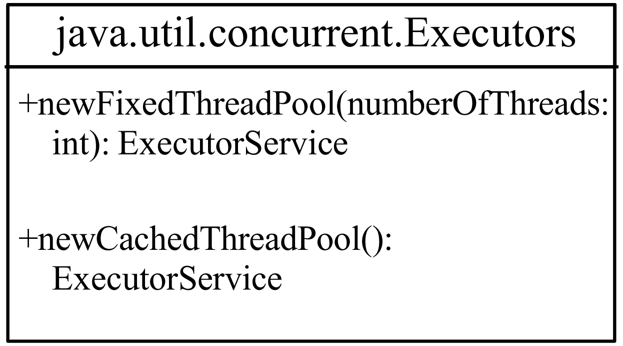
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shutdown.

Returns true if all tasks in the pool are terminated.

Creating Executors

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



java.util.concurrent.Executors

+newFixedThreadPool(numberOfThreads: int): ExecutorService

+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.

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);

Synchronization

- Synchronization is prevent data corruption
- Synchronization allows only one thread to perform an operation on a object at a time.
- If multiple threads require an access to an object, synchronization helps in maintaining consistency.

Example

```
public class Counter{
   private int count = 0;
            public int getCount(){
                    return count;
            }
            public void setCount(int count){
                    this.count = count;
            }
}
```

- In this example, the counter tells how many an access has been made.
- If a thread is accessing setCount and updating count and another thread is accessing getCount at the same time, there will be inconsistency in the value of count.

Fixing the example

```
public class Counter{
    private int count = 0;
   public synchronized int getCount(){
               return count;
   }

   public synchoronized void setCount(int count){
                this.count = count;
   }
}
```

- By adding the synchronized keyword we make sure that when one thread is in the setCount method the other threads are all in waiting state.
- The synchronized keyword places a lock on the object, and hence locks all the other methods which have the keyword synchronized. The lock does not lock the methods without the keyword synchronized and hence they are open to access by other threads.
- Note that constructors cannot be synchronized — using the synchronizedkeyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Object locking

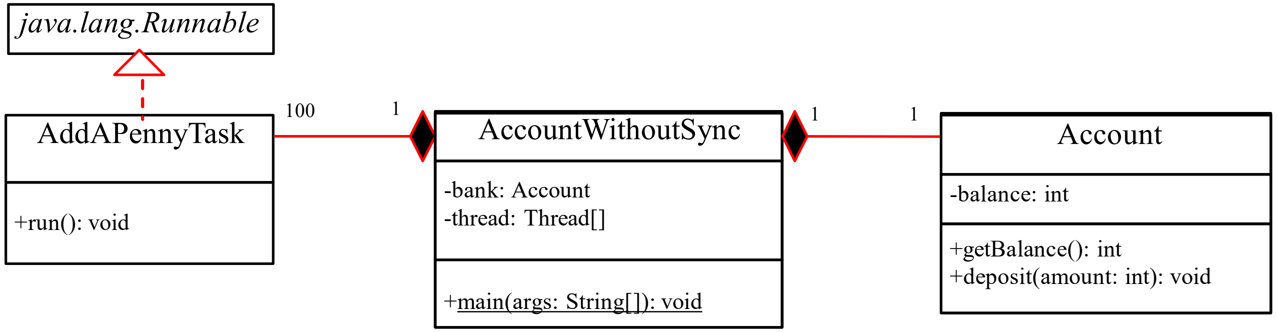
The object can be explicitly locked in this way

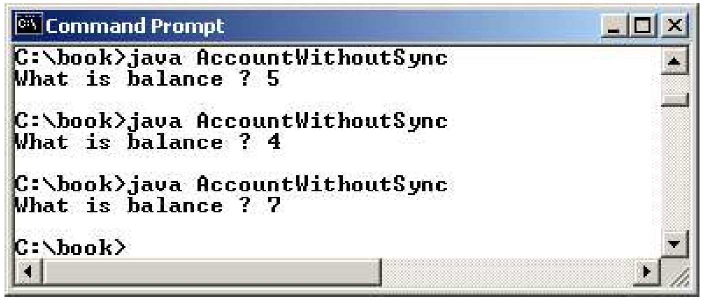
```
synchronized(myInstance){
try{
    wait();
}   catch(InterruptedException ex){
}
    System.out.println(“Iam in this “);
    notifyAll();
}
```

- The synchronized keyword locks the object. The wait keyword waits for the lock to be acquired, if the object was already locked by another thread. Notifyall() notifies other threads that the lock is about to be released by the current thread.
- Another method notify() is available for use, which wakes up only the next thread which is in queue for the object, notifyall() wakes up all the threads and transfers the lock to another thread having the highest priority.

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

Run

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.

Synchronized keyword

Problem: race conditions

Solution: give exclusive access to one thread at a time to code that manipulates a shared object.

- 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.
- The critical region is the entire deposit method.
- One way to correct the problem is to make Account thread-safe by adding the synchronized keyword in deposit:

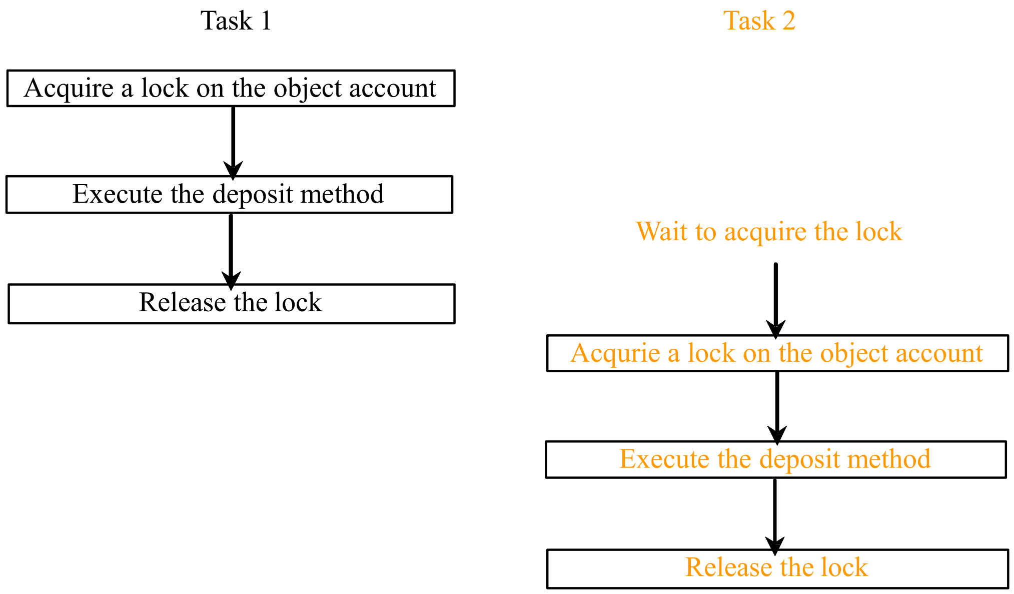
```
public synchronized void deposit(double amount)
```

Synchronizing Instance Methods and Static 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 Instance Methods and Static Methods

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



Task 1

Acquire a lock on the object account

Execute the deposit method

Release the lock

Task 2

Wait to acquire the lock

Acqurie a lock on the object account

Execute the deposit method

Release the lock

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:

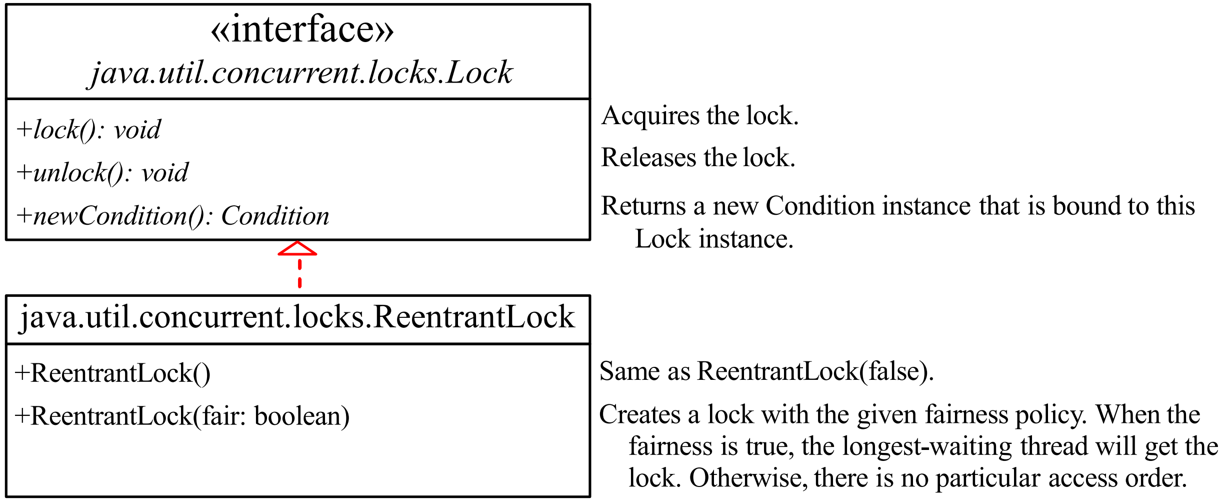
```
public synchronized void xMethod() {
  // method body
}
```

This method is equivalent to

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

Synchronization Using Locks

- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.
- You can use locks explicitly to obtain more control for coordinating threads.
- A lock is an instance of the <u>Lock</u> interface, which declares the methods for acquiring and releasing locks.
- <u>newCondition()</u> method creates <u>Condition</u> 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.

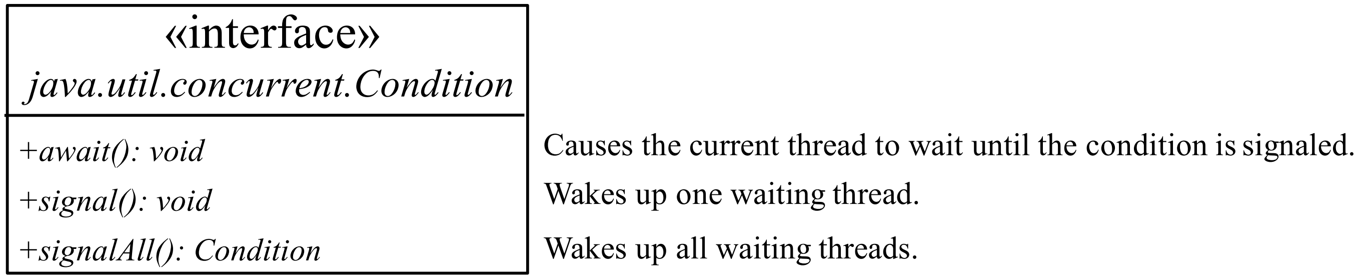
- False fairness policies grant a lock to a waiting thread without any access order.

- 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.

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.

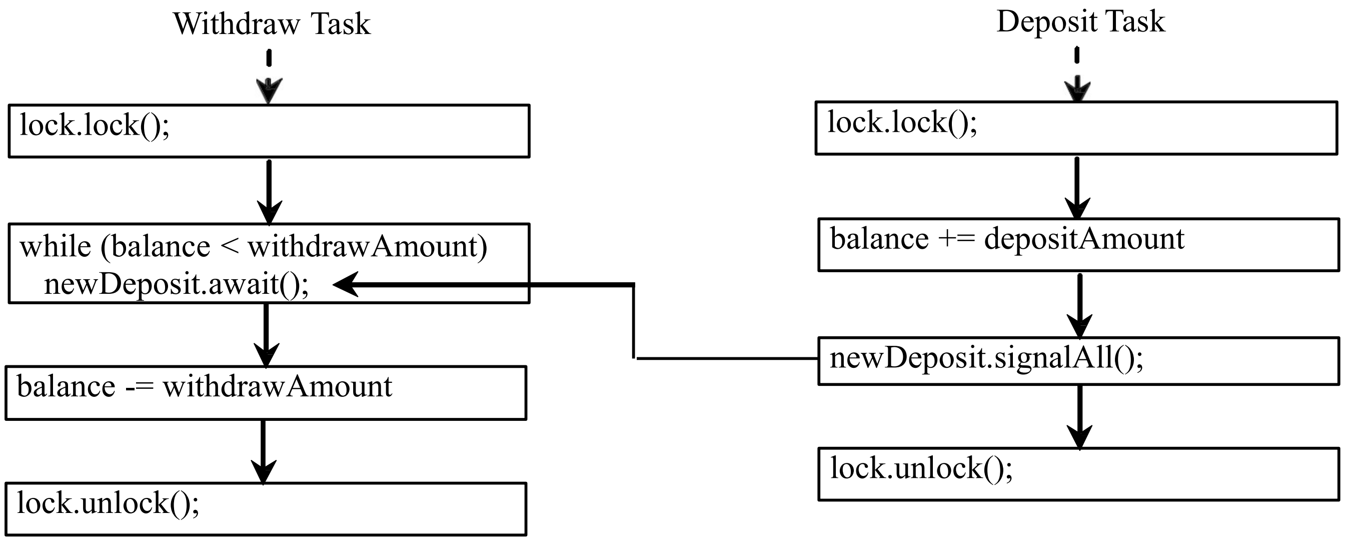
+signal(): void Wakes up one waiting thread.

+signalAll(): Condition Wakes up all waiting threads.

Cooperation Among Threads

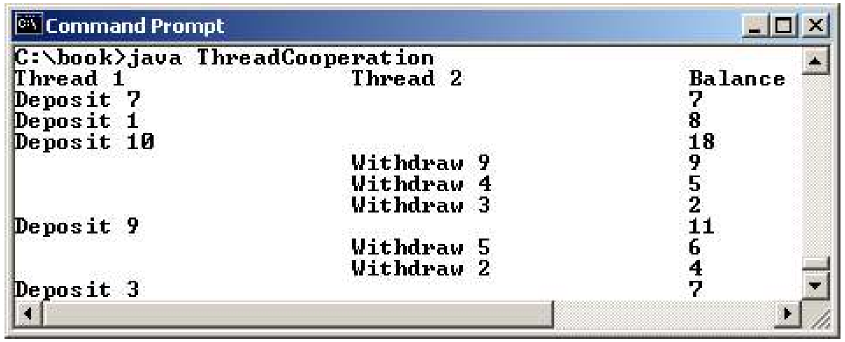
- Lock with a condition to synchronize operations: <u>newDeposit</u>
- If the balance is less than the amount to be withdrawn, the withdraw task will wait for the <u>newDeposit</u> condition.
- When the deposit task adds money to the account, the task signals the waiting withdraw task to try again.

- Interaction between the two tasks:



Example: Thread Cooperation

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                                          7
Deposit 1                                          8
Deposit 10                                         18
                          Withdraw 9               9
                          Withdraw 4               5
                          Withdraw 3               2
Deposit 9                                          11
                          Withdraw 5               6
                          Withdraw 2               4
Deposit 3                                          7
```

ThreadCooperation

Run

Code.Hub

Java’s Built-in Monitors

- Locks and conditions are new 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.
- Only one thread can execute a method at a time in the monitor.
- A thread enters the monitor by acquiring a lock (synchronized 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.

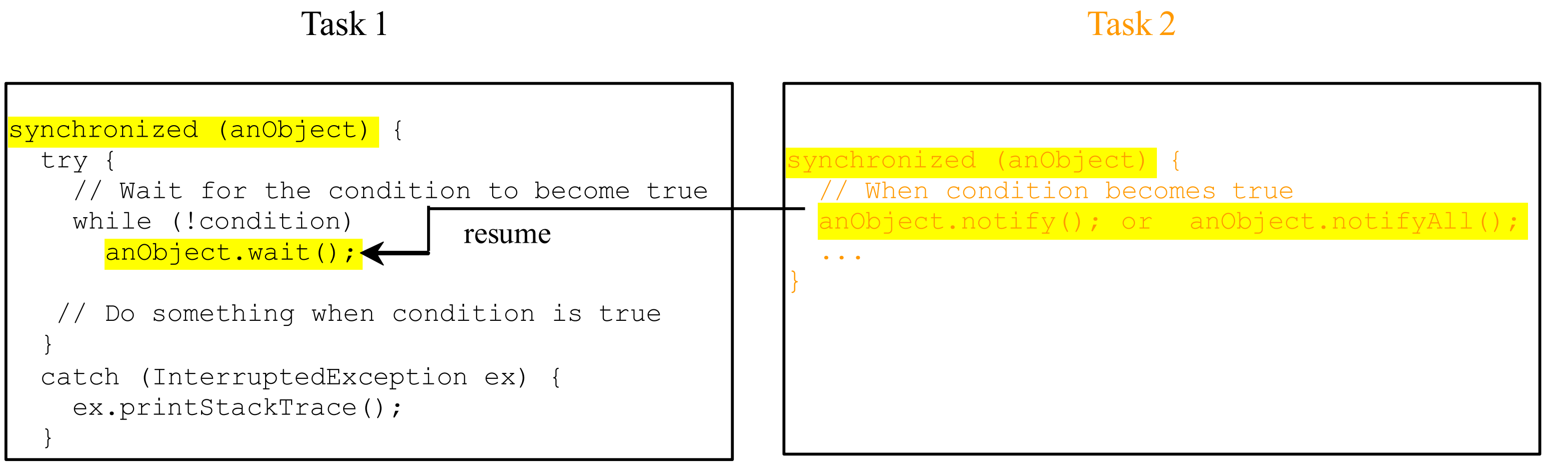
wait(), notify(), and notifyAll()

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();
  }
```

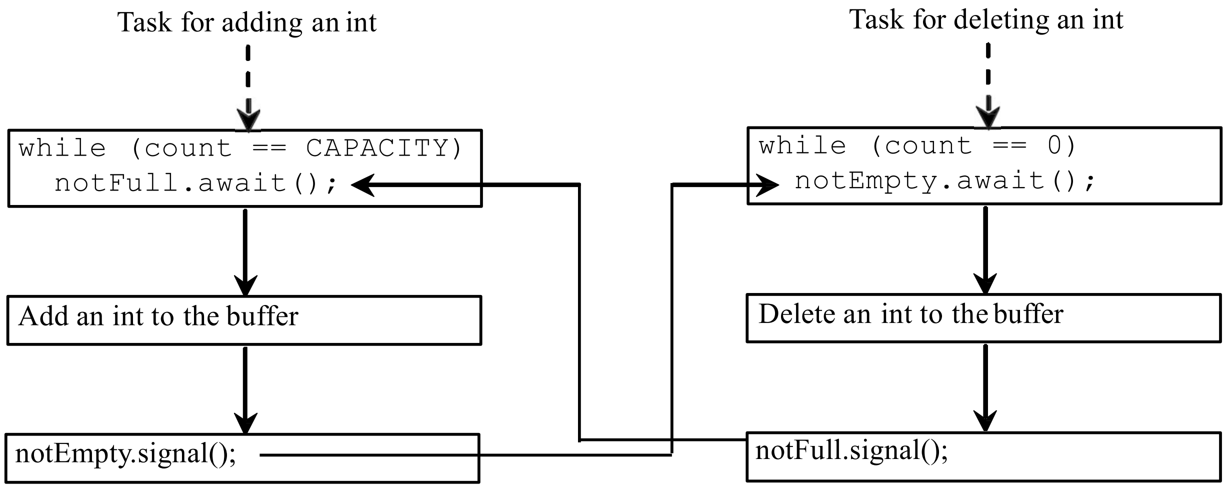
Task 2

```
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.

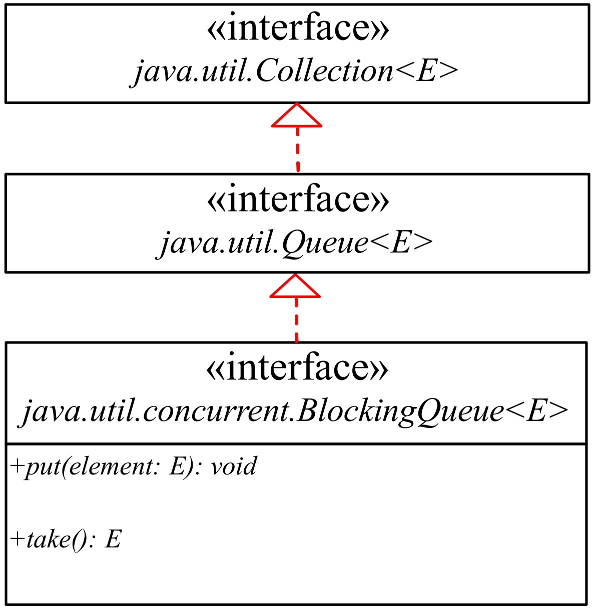
Case Study: Producer/Consumer

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. The interaction between the two tasks is shown



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.



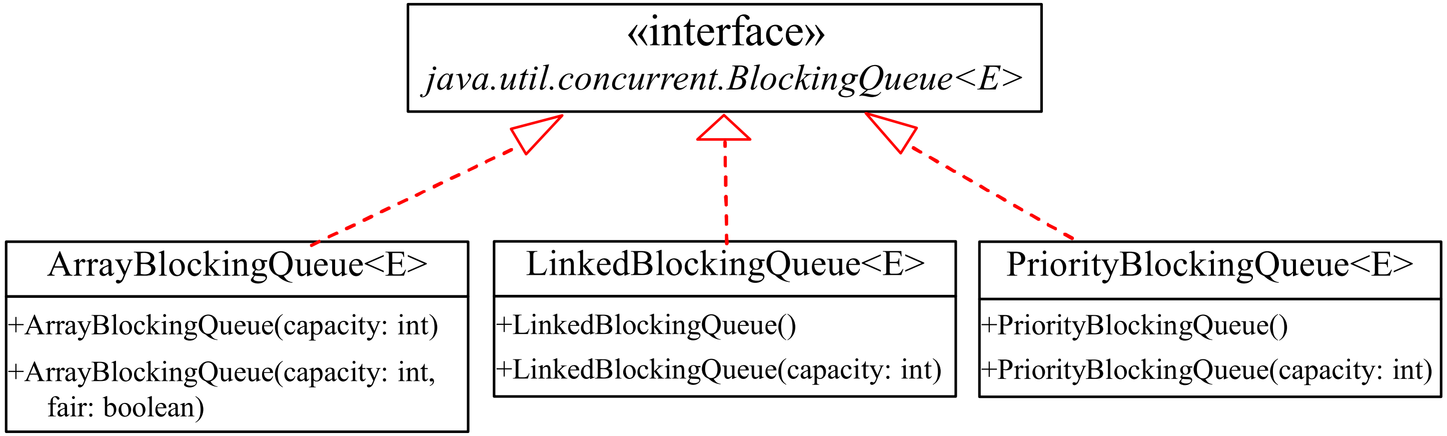
Inserts an element to the tail of the queue. Waits if the queue is full.

Retrieves and removes the head of this queue. Waits if the queue is empty.

Concrete Blocking Queues

Three concrete blocking queues ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are supported in JDK. All 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 unbounded or bounded priority queue.



«interface»
java.util.concurrent.BlockingQueue<E>

ArrayBlockingQueue<E>

+ArrayBlockingQueue(capacity: int)
+ArrayBlockingQueue(capacity: int, fair: boolean)

LinkedBlockingQueue<E>

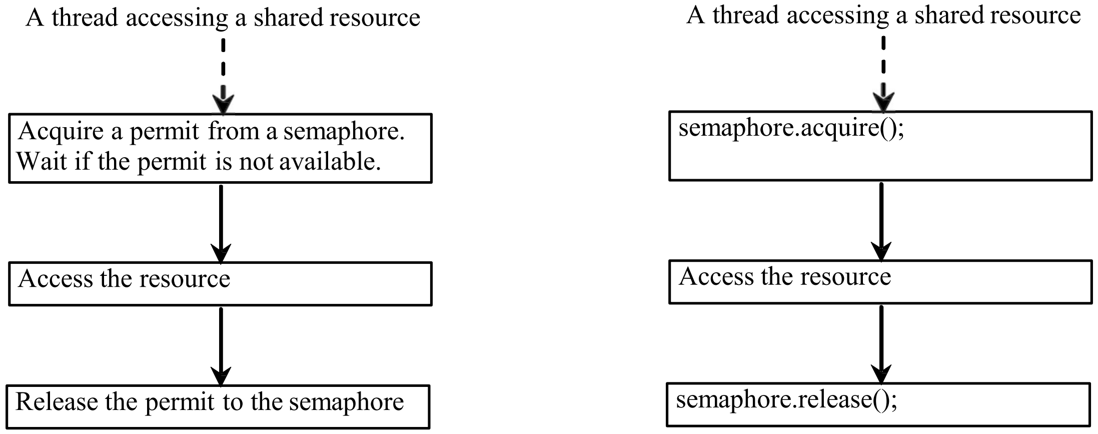
+LinkedBlockingQueue()
+LinkedBlockingQueue(capacity: int)

PriorityBlockingQueue<E>

+PriorityBlockingQueue()
+PriorityBlockingQueue(capacity: int)

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, as shown in Figure



A thread accessing a shared resource

Acquire a permit from a semaphore.
Wait if the permit is not available.

Access the resource

Release the permit to the semaphore

A thread accessing a shared resource

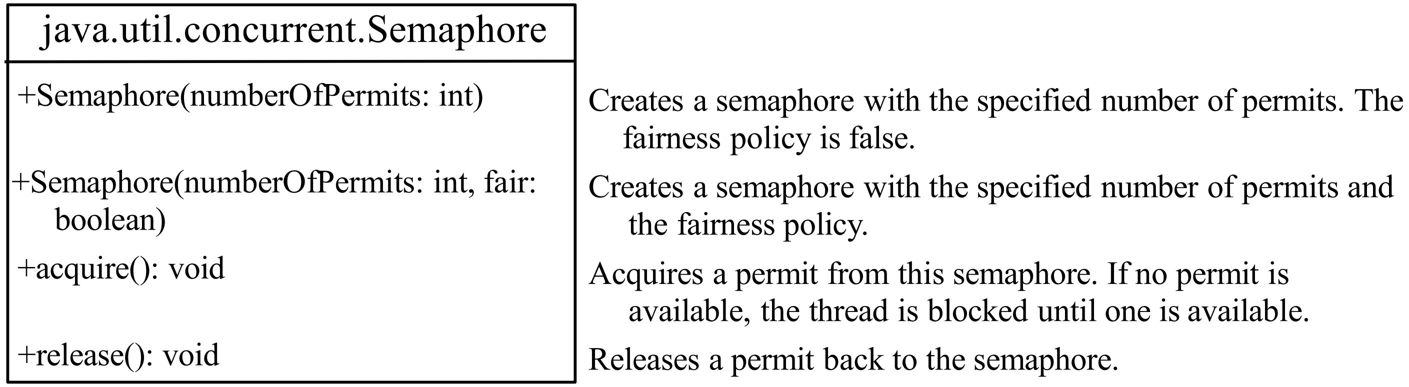
semaphore.acquire();

Access the resource

semaphore.release();

Creating Semaphores

To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in Figure. 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.

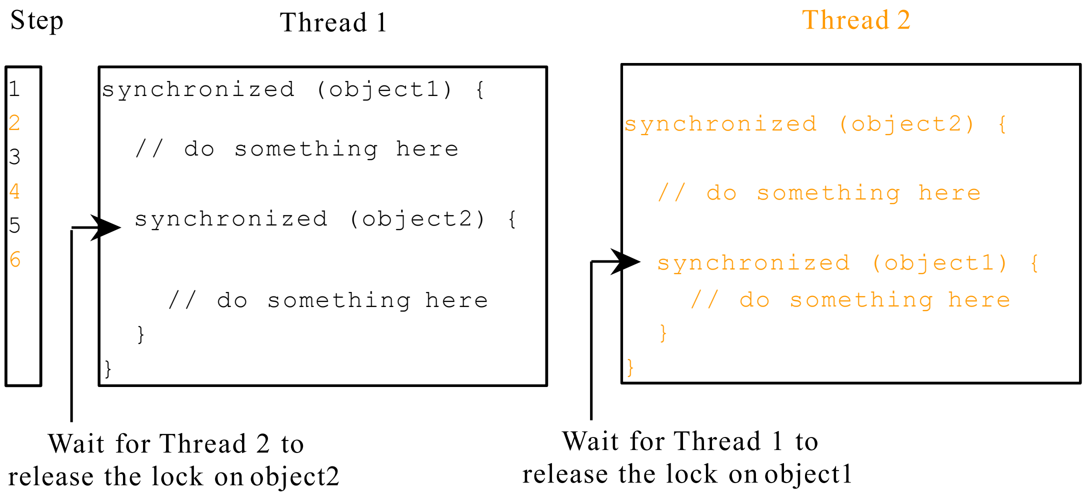


java.util.concurrent.Semaphore

+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.



Thread 1

```
synchronized (object1) {

  // do something here

  synchronized (object2) {

    // do something here
  }
}
```

Thread 2

```
synchronized (object2) {

  // do something here

  synchronized (object1) {
    // do something here
  }
}
```

Wait for Thread 2 to release the lock on object2

Wait for Thread 1 to release the lock on object1

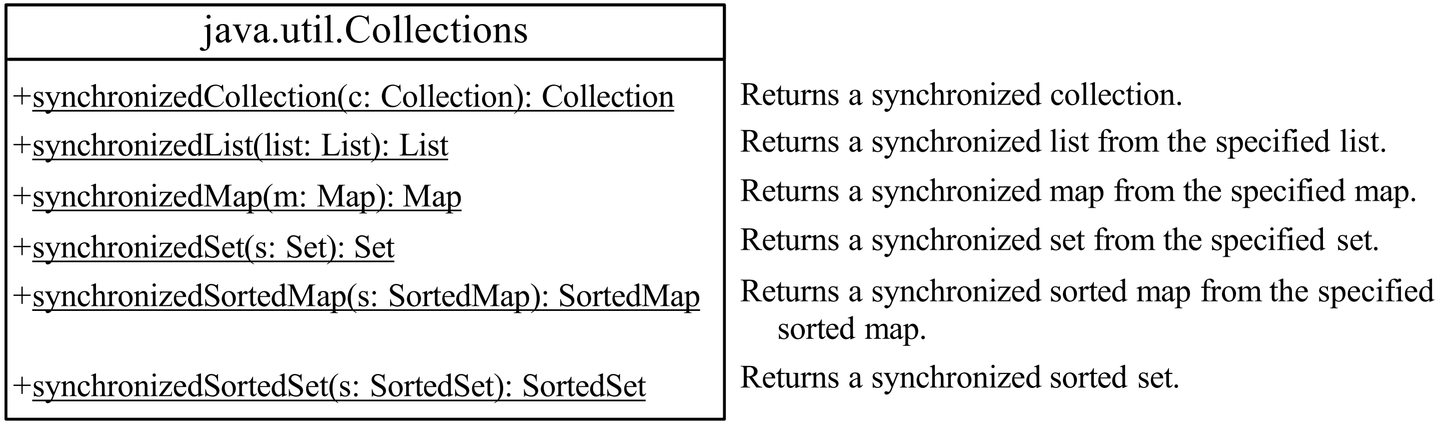
Preventing Deadlock

- Deadlock can be easily avoided by 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.
- How does this prevent deadlock in the previous example?

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 Collections class provides six static methods for creating *synchronization wrappers*.



java.util.Collections

+synchronizedCollection(c: Collection): Collection — Returns a synchronized collection.

+synchronizedList(list: List): List — Returns a synchronized list from the specified list.

+synchronizedMap(m: Map): Map — Returns a synchronized map from the specified map.

+synchronizedSet(s: Set): Set — Returns a synchronized set from the specified set.

+synchronizedSortedMap(s: SortedMap): SortedMap — Returns a synchronized sorted map from the specified sorted map.

+synchronizedSortedSet(s: SortedSet): SortedSet — Returns a synchronized sorted set.

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 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.

invokeLater and invokeAndWait methods

- In certain situations, you need to run the code in the event dispatcher thread to avoid possible deadlock. You can use the static methods, invokeLater and invokeAndWait, in the javax.swing.SwingUtilities class to run the code in the event dispatcher thread.
- You must put this code in the run method of a Runnable object and specify the Runnable object as the argument to invokeLater and invokeAndWait.
- The invokeLater method returns immediately, without waiting for the event dispatcher thread to execute the code.
- The invokeAndWait method is just like invokeLater, except that invokeAndWait doesn't return until the event-dispatching thread has executed the specified code.

Further Reading

- http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html
- http://www.deitel.com/articles/java_tutorials/20051126/JavaMultithreading_Tutorial_Part1.html

- Definition of threads
- Thread synchronization / communication between threads
- Thread-safe collections
- Callables and futures
- Threadpools and executors

Definition of threads

Thread synchronization / communication between threads

Thread-safe collections

Callables and futures

Threadpools and executors