EE3206
Java Programming and Applications

Lecture 8
Multithreading

Mr. Van Ting, Dept. of EE, CityU HK

Intended Learning Outcomes

- To understand the concept of concurrency.
- To understand the difference of a process and a thread.
- To define a thread using the Thread class and Runnable interface.
- To control threads with various Thread methods.
- To understand the life cycle of a thread.
- To execute tasks in a thread pool.
- To understand the cause of thread interference.
- To use synchronized methods to avoid race conditions.

Concurrency

- Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio.
- Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display.
- Software that can do such things is known as concurrent software.

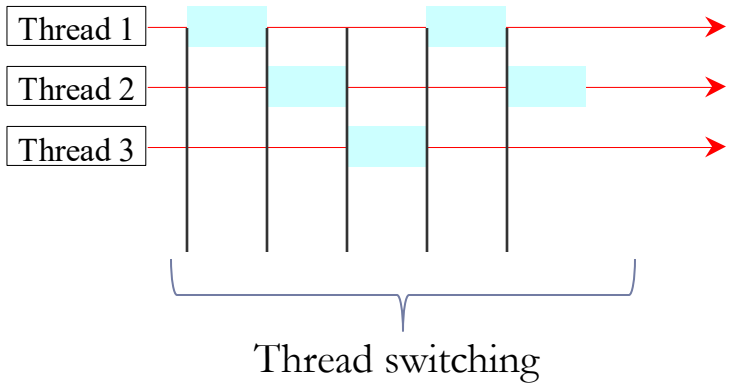
Processes and Threads

- A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its **own memory space**.
- Threads are sometimes called **lightweight processes**. Both processes and threads provide an execution environment, but threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic.
- Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread, called the **main thread**. This thread has the ability to create additional threads.

Thread Scheduling

- A thread is a thread of control flow of execution in a program. The JVM allows an application to have multiple threads of execution running in a pseudo-parallel fashion by thread switching.
- *Preemptive* scheduling - scheduling is based on time slices and priority.
- *Cooperative* scheduling - switch to another thread only when the current thread is blocked, or it yields the control voluntarily.
- The exact order of execution of different threads is *nondeterministic*, i.e. the execution order cannot be determined from the source code.

Multiple threads sharing a single CPU



CPU time is shared by multiple threads (switched very rapidly by OS) to give users a feel that they are running at the same time.

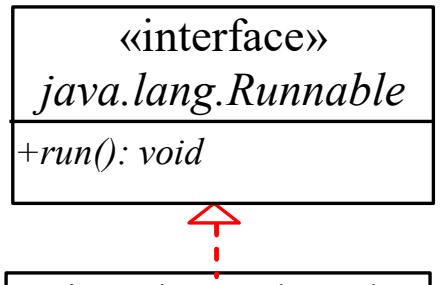
ThreadSwitching

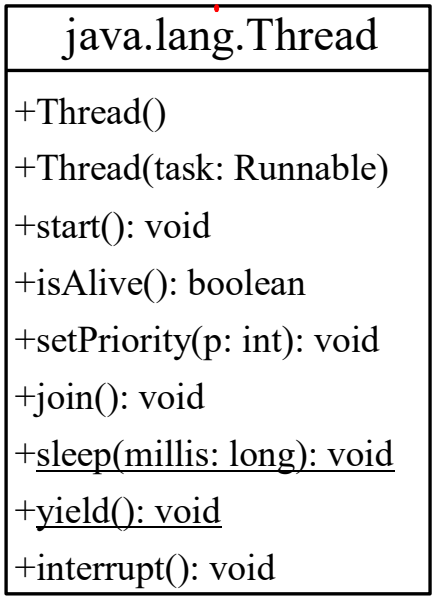
Pros and Cons of Using Multithreads

- Pros:
  - It is suitable for developing *reactive systems*, which continuously monitor arrays of sensors and react to control systems according to the sensor readings.
  - It makes applications more responsive to user input (e.g. games and animations).
  - It allows a server to handle multiple clients.
  - It may take advantage of the availability of multiple processor cores by executing the threads on different cores in parallel. For example, the Intel i7 CPU has 4-6 cores and supports hyper-threading. A programmer can divide the computation tasks into multiple threads to utilize the CPU cores.
- Cons:
  - Interaction and cooperation among different threads may lead to *safety* and *liveness* problems.
  - Multithreaded programs involve certain overhead, owing to the cost of thread creation, scheduling, and synchronization.

Thread Class

- The Thread class defines a number of methods useful for thread management.





Creates a default 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 the runnable object to sleep for a specified time in milliseconds.

Causes this thread to temporarily pause and allow other threads to execute.

Interrupts this thread.

Defining and Starting a Thread

- An application that creates an instance of Thread must provide the code that will run in that thread (i.e. the job you want to do). There are two ways to do this:
  1. Provide a ***Runnable*** object. The Runnable interface defines a single method, run(), meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:
  2. Subclass ***Thread***. The Thread class itself implements Runnable, though its run() method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        Runnable task = new HelloRunnable();
        Thread t = new Thread(task);
        t.start();
    }
}
```

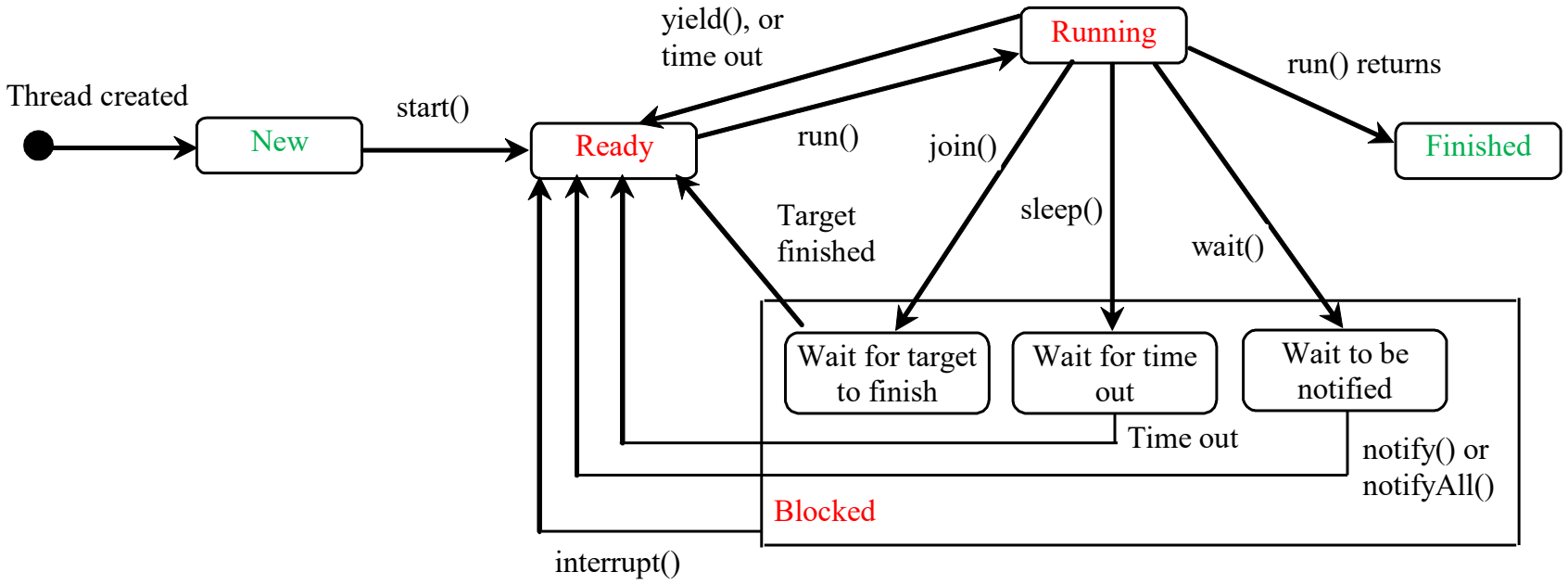
HelloRunnable

```
public class HelloThread extends Thread {

    @Override
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        Thread t = new HelloThread();
        t.start();
    }
}
```

HelloThread

Thread States

- A thread can be in one of five states: *New*, *Ready*, *Running*, *Blocked*, or *Finished*.
- The isAlive() method is used to test 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 in *New* or *Finished* state.



Pausing Execution with Sleep/Yield

- Thread.sleep(long mills) causes the current thread to suspend execution for a specified time in milliseconds.
- This is an efficient mean of making processor time available to the other threads of an application or other applications that might be running on a computer system.
- The sleep method throws **InterruptedException** when the sleeping thread is being interrupted.
- Another method Thread.yield() has similar effect that causes the currently executing thread to temporarily pause (with unspecified time) and allow other threads to execute.

```
public class SleepDemo {

   public static void main(String args[]) throws InterruptedException {
      System.out.print("Thinking");
      for(int i=0; i<20; i++) {
         Thread.sleep(1000);      //Pause for 1 seconds
         System.out.print(".");
      }
      System.out.println("Done!");
   }
}
```

SleepDemo

Queuing Execution with Join

- The join() method allows one thread to wait for the completion of another. If *t* is a Thread object whose thread is currently executing, invoking *t.join()* causes the current thread to pause execution until thread *t* terminates.
- Like sleep, join responds to an interrupt by resuming with an **InterruptedException**.

```
public class JoinDemo implements Runnable {

    public void run() {
        for(int i=0; i<100; i++)
            System.out.print("T");
    }

    public static void main(String args[]) throws InterruptedException {
        Thread t = new Thread(new JoinDemo());
        t.start();
        t.join();       // main thread waits for the finish of thread t

        for(int i=0; i<100; i++)
            System.out.print("M");
    }
}
```

JoinDemo

Stopping Execution with Interrupt

- An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.
- Thread A sends an interrupt to thread B by invoking interrupt() on B (i.e. the thread to be interrupted). The interrupt() method interrupts B in one of the following two ways:

1. If B is in **Ready or Running** state:
   - B's **interrupt status flag** will set. Thread B should periodically invoke Thread.interrupted() to check the flag and respond to any interrupt arisen.

InterruptDemo1

2. If B is in **Blocked** state:
   - B wakes up and receives an **InterruptedException**. Thread B must catch the **InterruptedException** and handle it accordingly.

InterruptDemo2

Thread Priority

- Every thread has a priority. Threads with higher priority are executed in preference to threads with lower 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 // 1
  - Thread.NORM_PRIORITY // 5
  - Thread.MAX_PRIORITY // 10

Thread Pool

- A thread "dies" when it finishes the given task. The thread object remains in memory but cannot be reused for execution anymore. You therefore need to create a new thread object for every task to execute.
- **Thread pool** consists of **worker threads** which will return to the pool for reuse when they finish the job. Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.
- **ExecutorService** is an interface for executing and managing thread tasks
- **Executors** is a factory class for providing implementations of **ExecutorService**
- For example:

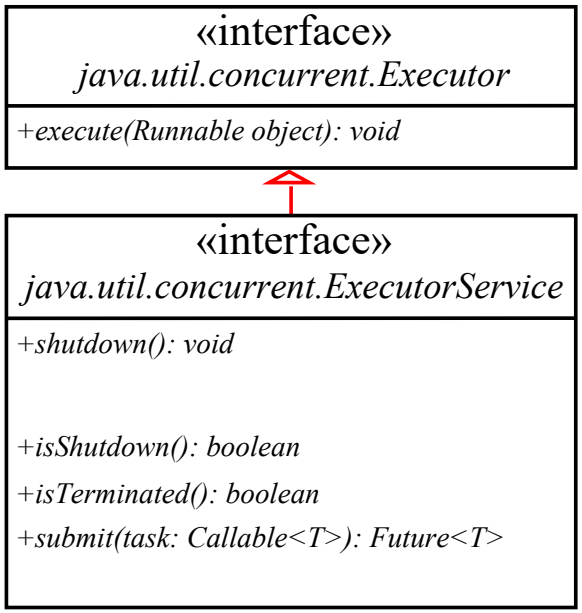
```
// creates a thread pool with a fixed number of threads executing concurrently
ExecutorService pool = Executors.newFixedThreadPool(numOfThreads);
// creates a thread pool that creates new threads as needed
ExecutorService pool = Executors.newCachedThreadPool();
//use a pool to execute a Runnable task
pool.execute(task);
```

int

Runnable

ExecutorDemo

Thread Pool



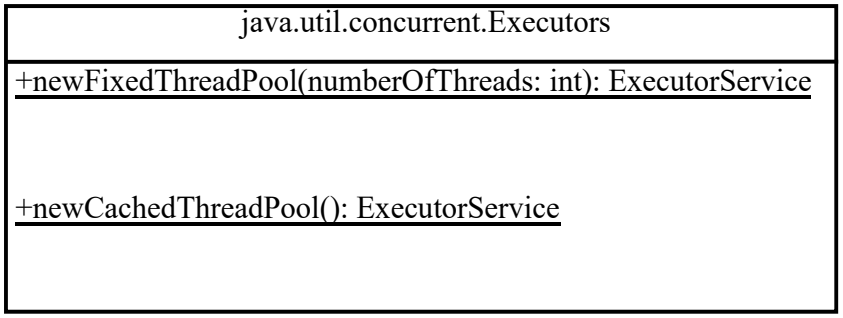
Executes the runnable task.

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

Returns true if the executor has been shutdown.

Returns true if all tasks in the pool are terminated.

Submits a value-returning task for execution and returns a Future representing the pending results of the task.



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. Idle threads are kept for 60 seconds.

Callable and Future

- The **void run()** method in the Runnable interface has no return value.
- If the thread is required to return a result, you can use the **Callable<V>** interface and the **Future<T>** interface to help you manage thread-cooperation.
  - The Callable<V> interface has a method **V call()** that returns a result of type V.
  - You execute (submit) a Callable<V> by an ExecutorService that manages a thread pool.
  - When a task is carried out by a Callable<V>, the result will be available in some future time.
  - The result produced by the Callable can be retrieved by the **V get()** method of the interface Future<V>.

MultiThreadMaxFinder

Fork/Join Framework

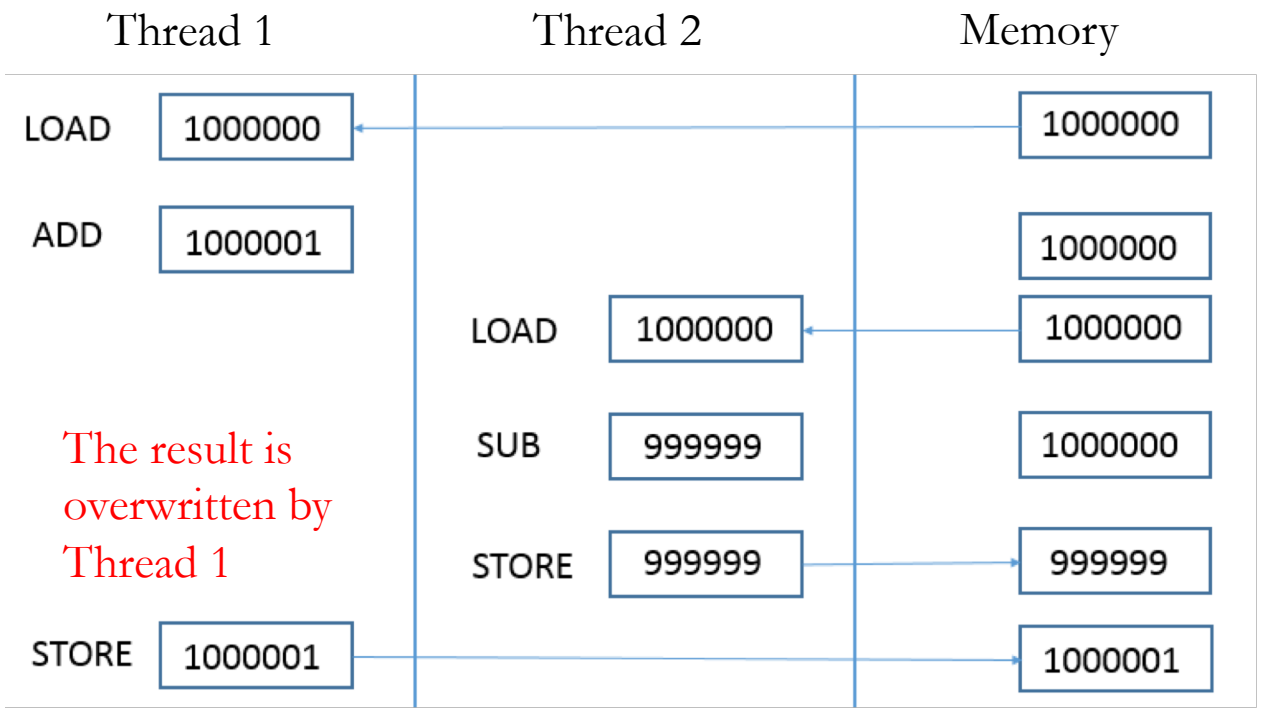
- The Fork/Join framework (introduced in Java 7) is suitable in situations where a task can be divided into multiple subtasks that can be executed in parallel.
- The fork/join framework creates a pool (**ForkJoinPool**) of threads to execute subtasks (**ForkJoinTask**). There are two related concrete subclasses of the abstract ForkJoinTask.
  - **RecursiveAction** – used to implement a task that does not yield a result
  - **RecursiveTask** – used to implement a task that yield a result
- When subtasks are finished, the partial results can be combined to get the final result. The typical logic of the compute() method is as follow.

```
if (task is small) {
   Solve the task directly.
}
else {
   Divide the task into subtasks.
   Launch the subtasks asynchronously (the fork stage).
   Wait for the subtasks to finish (the join stage).
   Combine the results of all subtasks.
}
```

ForkJoinTest

Race Condition (Thread Interference)

- Threads communicate primarily by sharing access to fields. This form of communication is extremely efficient but may make errors.
- The example below shows two different threads add one to and subtract one from the cache. This problem is known as Race Condition/Hazard.



AccountWithoutSync

RaceCondition

Atomicity

- An action is atomic (indivisible) if it appears to the rest of the system to occur at once without being interrupted.
- Atomic variables (java.util.concurrent.atomic) are introduced in Java 5 to support thread safety for the sharing of simple wrapper class objects.
- Actions performed on an atomic variable are not divisible - it either happens completely or not at all.
- In Java, read and write actions on an object's field of any type, except long and double, are atomic.
  - int, short, float, char, boolean, object reference are 16-bit or 32-bit entities.
  - long and double are 64-bit entities. The CPU needs multiple memory accesses to transfer 64 bits to/from main memory.
- If a field of type long or double is declared **volatile**, read and write on that field are also guaranteed to be atomic.

Synchronization

- An important safety property of threads is the consistency of shared objects.
- 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 **NonAtomicNum** class is **not thread-safe**.
- The common resource accessed by multiple threads are known as **critical region**. More than one thread must be prevented from simultaneously entering the critical region of a program.
- The tool needed to prevent thread interference is **synchronization**. To avoid resource conflicts, you can use the keyword synchronized to declare a synchronized method.
- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object are blocked (suspend execution) until the first thread is done with the object.
- The critical region in the previous example includes the increment and decrement methods. You can add the keyword **synchronized** to its method header. As a result, only one thread can access the method at a time, and hence **NonAtomicNum** becomes **thread-safe**.

```
public synchronized void deposit(double amount) {
        // method body
}
```

AccountWithSync

Monitor (Intrinsic Lock)

- A *monitor (or intrinsic lock)* is an object with mutual exclusion and synchronization capabilities.
  - Every class/object has an intrinsic lock associated with it.
  - Invoking a synchronized instance method of an object acquires the intrinsic lock on the object, and invoking a synchronized static method of a class acquires the lock on the class.

- *Any object can (provide) 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.
  - 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.
  - 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.

Synchronizing Statements

- A synchronized statement can be used to acquire a lock on any object, not just the *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 vs. Methods

- Synchronizing statement can be used to synchronize just a block of code (with better performance by reducing blocking) instead of the whole method.
- 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
  }
}
```

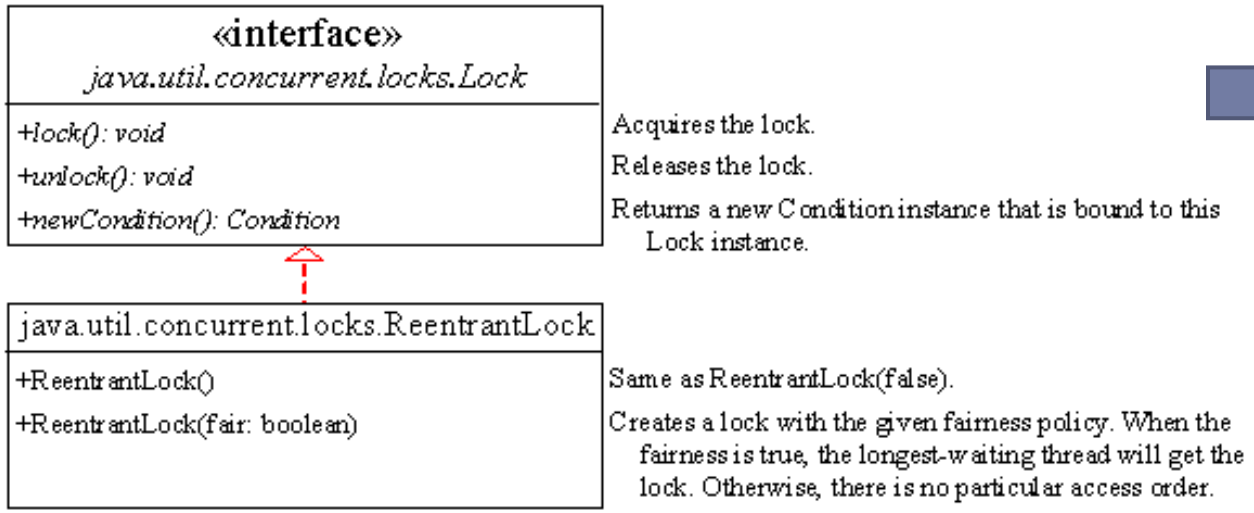
Synchronization on Multiple Locks

- Class *MsLunch* has two instance fields, c1 and c2, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking.
- Instead of using synchronized methods, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Reentrant Synchronization

- A thread cannot acquire a lock owned by another thread, but a thread can acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables reentrant synchronization.
- This describes a situation where synchronized code, directly or indirectly, invokes a method (e.g. recursion) that also contains synchronized code, and both sets of code use the same lock.
- A re-entrant lock keeps a *hold count* that keeps track of the nested calls to the lock method.
- A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread coordination/communication.



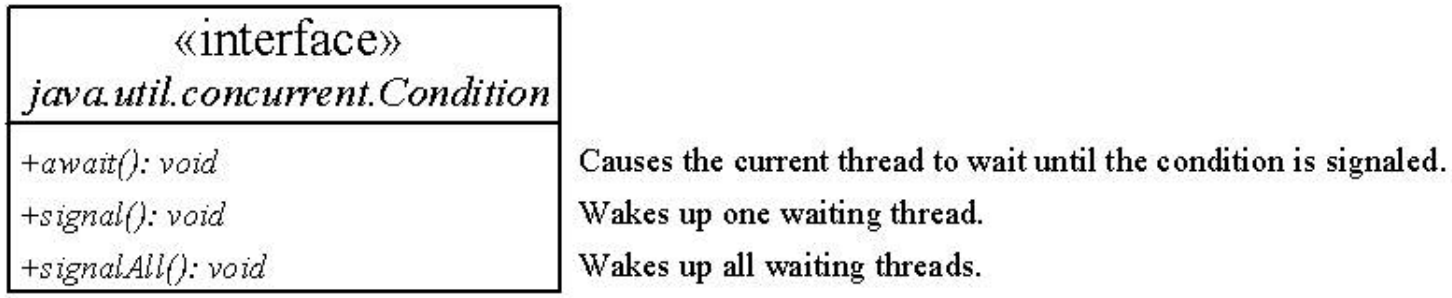
AccountWithSyncUsingLock

Fairness Policy

- ReentrantLock is a concrete implementation of Lock interface for creating mutual 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 (default) fairness policies grant a lock to a waiting thread without any access order.
- Programs using fair locks accessed by many threads may have worse overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Conditions on Lock

- Very often, a thread enters a critical region, only to discover that it can't proceed until a condition is fulfilled. Conditions provide a means for one thread to suspend its execution (to wait) until notified by another thread that some state condition may now be true.
- 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 lock associated with this Condition to be released and the current thread to wait until the Condition is signaled. The lock will then be re-acquired again before the thread resumes from the await() call.
  - The signal()/signalAll() method wakes up one/all waiting thread(s).



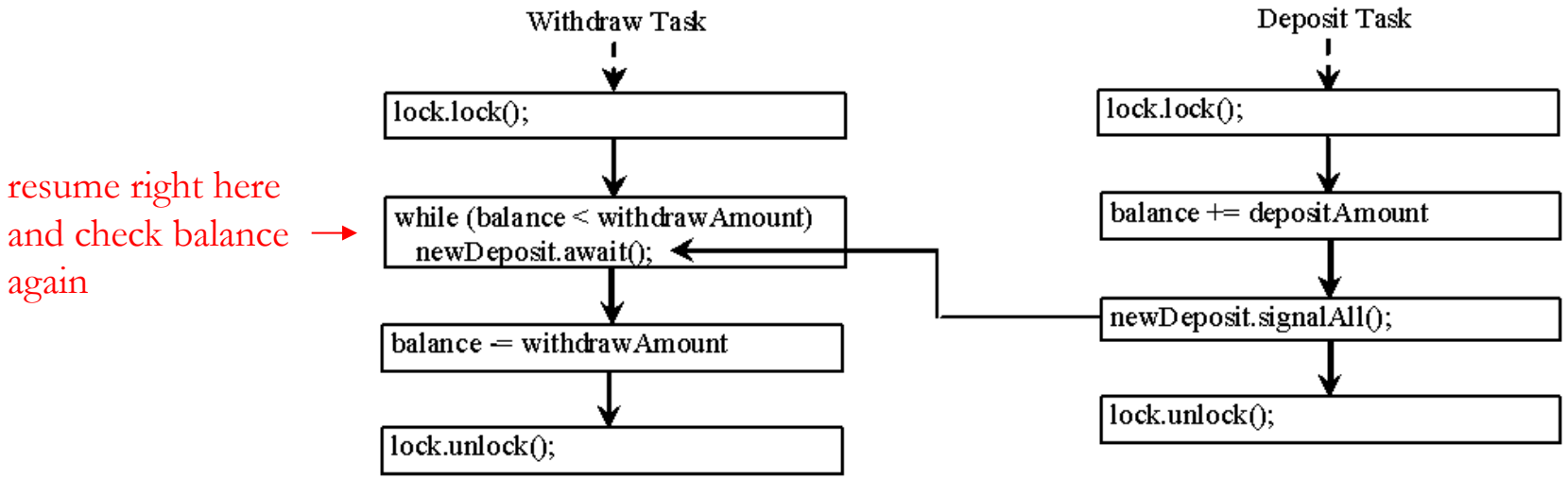
Producer and Consumer Problem

- 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.

ProducerConsumer

Thread Cooperation

- The preceding problem can be solved by using 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 (and release the lock associated with it).
- When the deposit task adds money to the account, the task signals the waiting withdraw tasks to try again. The withdraw tasks must re-acquire the lock before the await() method returns.
- The interaction between the two tasks is shown in the figure below.

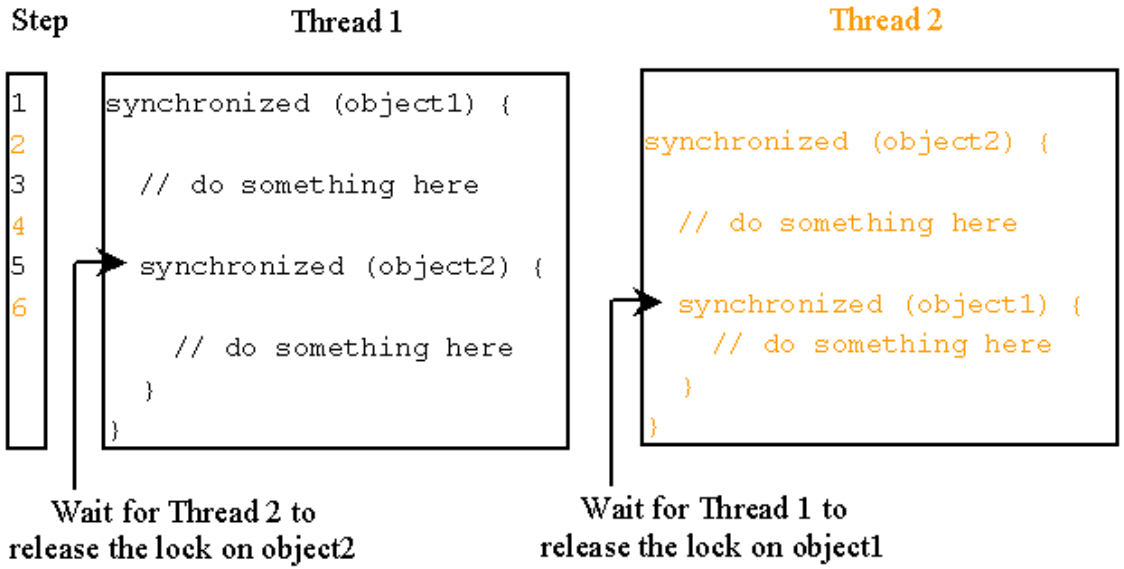


Liveness Problems

- A concurrent application's ability to execute in a timely manner is known as its *liveness*.
  - *Deadlock*, the most common liveness problem, describes a situation where two or more threads are blocked forever, waiting for each other.
  - *Livelock* describes a situation in which two or more threads continuously change their states in response to changes in the other threads without doing any useful work (*they are not blocked!*).
    - Ex. When two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.
  - *Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.
    - Ex. Suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Deadlock Example

- 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, as shown in this figure. 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 locks, 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 previous 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.