Advanced Programming
CSE 201
Instructor: Sambuddho

(Semester: Monsoon 2025)
Week 8 - Threads and Multithreaded Programming

Overview

- *Threads - sub-units of processes that are ``lightweight'', i.e. share lot of memory locations. They run within one program.*
- *OS schedules → JVM → (schedules) → Processes/java program → uses thread library to schedule → threads.*

- *In most OSes, threads are not very different from processes.*
- *In java they are different!*

Create a Thread

```
public interface Runnable
{
   void run();
}
```

- *1. Implement run() of Runnable interface.*

```
Runnable r = () -> { task code };
```

- *2. Construct a Thread object from the Runnable.*

```
var t = new Thread(r);
```

- *3. Start the thread.*

```
t.start();
```

- *Alternatively:*

```
class MyThread extends Thread
{
   public void run()
   {
      task code
   }
}
```

compact1, compact2, compact3
java.lang

Class Thread

java.lang.Object
    java.lang.Thread

All Implemented Interfaces:

Runnable

Direct Known Subclasses:

ForkJoinWorkerThread

Create a Thread - Example

```
1  package threads;
2
3  /**
4   * @version 1.30 2004-08-01
5   * @author Cay Horstmann
6   */
7  public class ThreadTest
8  {
9     public static final int DELAY = 10;
10    public static final int STEPS = 100;
11    public static final double MAX_AMOUNT = 1000;
12
13    public static void main(String[] args)
14    {
15       var bank = new Bank(4, 100000);
16       Runnable task1 = () ->
17       {
18          try
19          {
20             for (int i = 0; i < STEPS; i++)
21             {
22                double amount = MAX_AMOUNT * Math.random();
23                bank.transfer(0, 1, amount);
24                Thread.sleep((int) (DELAY * Math.random()));
25             }
26          }
27          catch (InterruptedException e)
28          {
29          }
30       };
31
32       Runnable task2 = () ->
33       {
34          try
35          {
36             for (int i = 0; i < STEPS; i++)
37             {
38                double amount = MAX_AMOUNT * Math.random();
39                bank.transfer(2, 3, amount);
40                Thread.sleep((int) (DELAY * Math.random()));
41             }
42          }
43          catch (InterruptedException e)
44          {
45          }
46       };
47
48       new Thread(task1).start();
49       new Thread(task2).start();
50    }
51 }
```

java.lang.Thread 1.0

- `Thread(Runnable target)`

  constructs a new thread that calls the `run()` method of the specified target.

- `void start()`

  starts this thread, causing the `run()` method to be called. This method will return immediately. The new thread runs concurrently.

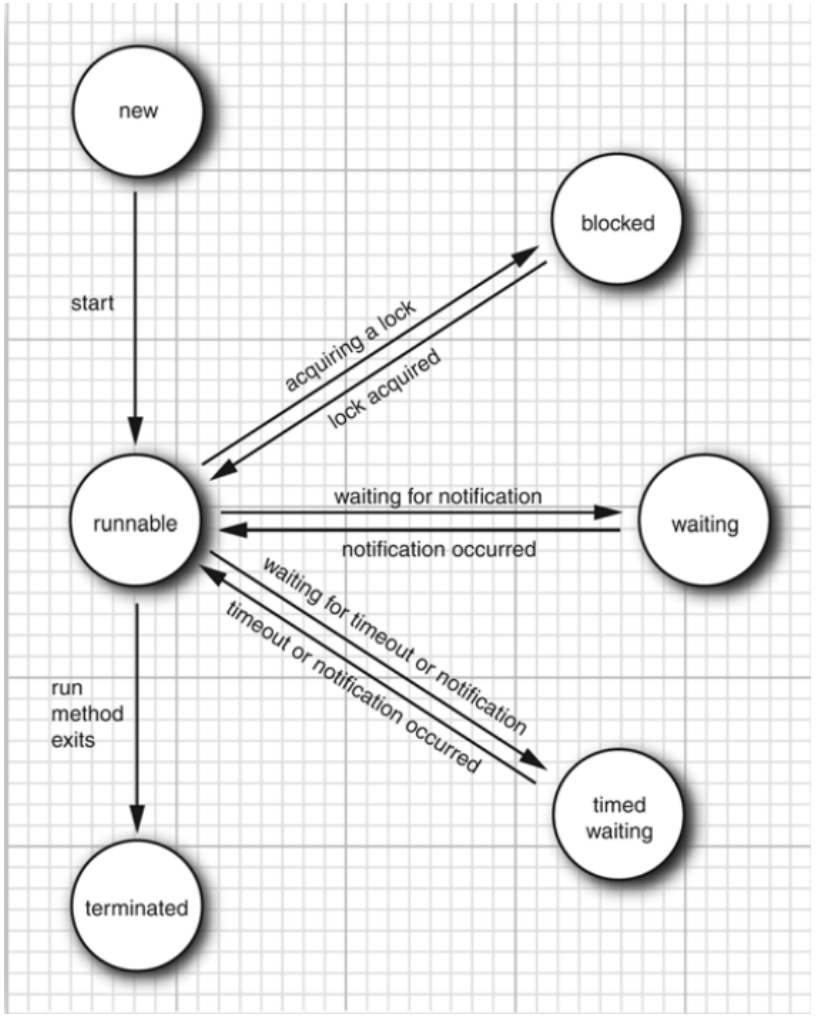
- `void run()`

  calls the `run` method of the associated `Runnable`.

- `static void sleep(long millis)`

  sleeps for the given number of milliseconds.

Thread States



- *You need to periodically yeild() a thread for others to be scheduled, because of non-premptive scheduling.*

Thread Termination

- *1. Happens due to two reasons - normal process termination or due to uncaught exception.*

java.lang.Thread 1.0

- `void join()`

  waits for the specified thread to terminate.

- `void join(long millis)`

  waits for the specified thread to die or for the specified number of milliseconds to pass.

- `Thread.State getState()` 5

  gets the state of this thread: one of `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, or `TERMINATED`.

- `void stop()`

  stops the thread. This method is deprecated.

- `void suspend()`

  suspends this thread's execution. This method is deprecated.

- `void resume()`

  resumes this thread. This method is only valid after `suspend()` has been invoked. This method is deprecated.

Thread Interruption

- *– Java thread are non-preemptive.*
- *– Grab a threads attention - interruption.*
  - *Process terminates.*
  - *Exception.*
  - *Interrupt occurred.*

interrupt

```
public void interrupt()
```

```
while (!Thread.currentThread().isInterrupted() && more work to do)
{
   do more work
}
```

  - *Blocked and sleeping threads cannot be interrupted. InterruptedException occurs.*
  - *Interruption doesn't mean termination - that is system dependent if that happens.*
  - *Only semantic - grab threads attention!*
  - *Thread can choose reaction to interrupt.*

Thread Interruption - Example of checking if thread was interrupted.

```
Runnable r = () -> {
   try
   {
      . . .
      while (!Thread.currentThread().isInterrupted() && more work to do)
      {
         do more work
      }
   }
   catch(InterruptedException e)
   {
      // thread was interrupted during sleep or wait
   }
   finally
   {
      cleanup, if required
   }
   // exiting the run method terminates the thread
};
```

Thread Interrupt Methods

java.lang.Thread 1.0

- `void interrupt()`

  sends an interrupt request to a thread. The interrupted status of the thread is set to `true`. If the thread is currently blocked by a call to `sleep`, then an `InterruptedException` is thrown.

- `static boolean interrupted()`

  tests whether the *current* thread (that is, the thread that is executing this instruction) has been interrupted. Note that this is a static method. The call has a side effect—it resets the interrupted status of the current thread to `false`.

- `boolean isInterrupted()`

  tests whether a thread has been interrupted. Unlike the `static interrupted` method, this call does not change the interrupted status of the thread.

- `static Thread currentThread()`

  returns the `Thread` object representing the currently executing thread.

Daemon Threads

- *Daemonizes a thread - sets it to a background thread and doesn't require joining back to parent thread.*

```
t.setDaemon(true);
```

java.lang.Thread 1.0

- `void setDaemon(boolean isDaemon)`

  marks this thread as a daemon thread or a user thread. This method must be called before the thread is started.

Thread Names

- *Thread can have names.*

```
var t = new Thread(runnable);
t.setName("Web crawler");
```

Handlers of Uncaught Exceptions

- *– run() method can die from unchecked exceptions.*
- *– There is no catch{} clause to which the exception can be transferred to.*
- *– Just before the thread dies the handler for the exception is called.*
- *– Handler must belong to a class that implements Thread.UncaughtExceptionHandler interface.*

- *void uncaughtException(Thread t, Throwable e);*

- *– Install handler using Thread.setUncaughtExceptionHandler() or using Thread.setDefaultUncaughtEceptionHandler().*
- *– Both of these could be used for ``graceful degradation.''*

-

Handlers of Uncaught Exceptions

- *ThreadGroup class implemenent the Thread.UncaughtExceptionHandler interface. Thread.uncaughtException() takes the following action.*
- *1. If thread group has a parent thread then call the uncaughtException of the parent group.*
- *2. Else, if the Thread.getDefaultUncaughtException != null then that the default exception handler is called.*
- *3. Else if the Throwable argument of uncaughtException() is an instance of ThreadDeath (regular thread termination), then nothing happens.*
- *4. Else, the thread name and stack trace is printed on System.err.*
-

Handlers of Uncaught Exceptions

java.lang.Thread 1.0

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()` 5

  sets or gets the default handler for uncaught exceptions.

- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` 5

  sets or gets the handler for uncaught exceptions. If no handler is installed, the thread group object is the handler.

java.lang.Thread.UncaughtExceptionHandler 5

- `void uncaughtException(Thread t, Throwable e)`

  defined to log a custom report when a thread is terminated with an uncaught exception.

Thread Priorities

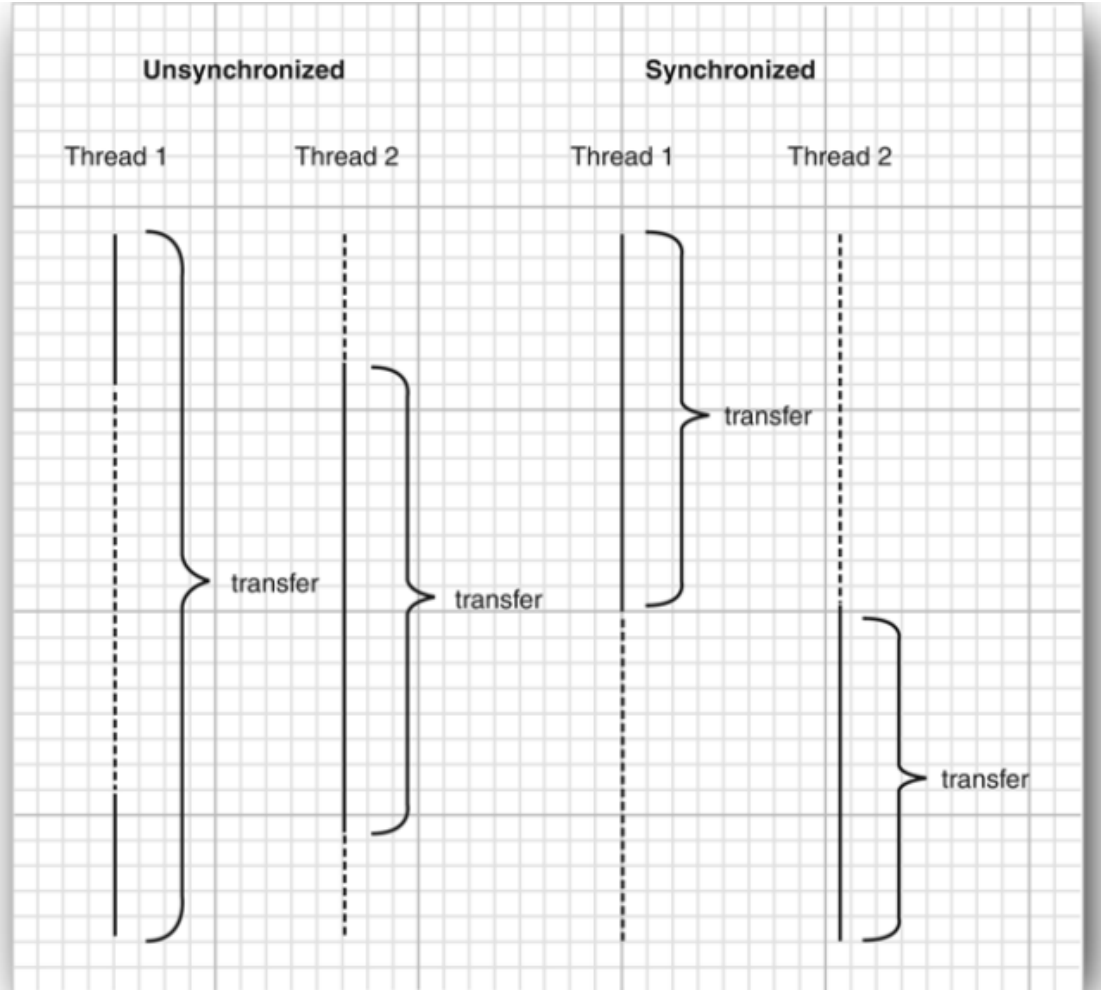
- *– Threads have priorities, determines which thread is chosen to run next by the library, after a process yeilds or is interrupted.*
- *– Default priority is the priority of the parent thread.*

- *Thread.setPriority(int newPriority)*

- *MIN_PRIORITY (1) < NOMR_PRIORITY (5) < MAX_PRIORITY(10).*

- *Priorities are implementation dependent.*

Thread Synchronization - Locks Basics



Synchronized/serialized access versus unsynchronized.

Basic ideas:

Race Condition: Two parallel threads access a common object and their order of update leaves the object in a undermined state.

Solution: Serialize access to the common object aka critical section.

Thread Synchronization - Locks Basics

ReentrantLock class used.

```
Lock mylock = new ReentrantLock();
mylock.lock();
try{
    Critical section
}
finally{
 mylock.unlock();
}
```

Thread Synchronization - Locks Basics

```
public class Bank
{
   private Lock bankLock = new ReentrantLock();
   . . .
   public void transfer(int from, int to, int amount)
   {
      bankLock.lock();
      try
      {
         System.out.print(Thread.currentThread());
         accounts[from] -= amount;
         System.out.printf(" %10.2f from %d to %d", amount, from, to);
         accounts[to] += amount;
         System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
      }
      finally
      {
         bankLock.unlock();
      }
   }
}
```

java.util.concurrent.locks.Lock 5

- `void lock()`

  acquires this lock; blocks if the lock is currently owned by another thread.

- `void unlock()`

  releases this lock.

java.util.concurrent.locks.ReentrantLock 5

- `ReentrantLock()`

  constructs a reentrant lock that can be used to protect a critical section.

- `ReentrantLock(boolean fair)`

  constructs a lock with the given fairness policy. A fair lock favors the thread that has been waiting for the longest time. However, this fairness guarantee can be a significant drag on performance. Therefore, by default, locks are not required to be fair.

Fair lock: Tries to prevents starvation.

Thread Synchronization - Regular vs Reentrant Locks

Regular locks: One thread holds the lock, the others attempting are blocked until the first one unlocks it; aka Mandatory Locks

Reentrant lock: More like advisory locks. A thread can acquire a lock multiple times but then it must also be locked the same number of times. Uses a count to ensure that the number of locks and unlocks are equal.

Thread-Safe Collections

- *Blocking queues - If multiple thread read from or write to blocking queues then one of the them blocks if it is attempting to read from an empty queue or write to a full queue.*

- *Useful for solving ``producer/consumer'' like problem scenarios.*

- *Several variations of queue add and remove methods - some take a timeout argument.*

- *Variants: LinkedBlockingQueue (unbounded, upper bound optional) , ArrayBlockingQueue (constructed with a fixed capacity), LinkedBlockingDequeue (double-ended version of LinkedBlockingQueue), PriorityBlockingQueue,DelayQueue,LinkedTransferQueue (allows producer to wait until consumer is ready).*

Thread-Safe Collections

- *Blocking Queue Method*

Method	Normal Action	Action in Special Circumstances
`add`	Adds an element	Throws an `IllegalStateException` if the queue is full
`element`	Returns the head element	Throws a `NoSuchElementException` if the queue is empty
`offer`	Adds an element and returns `true`	Returns `false` if the queue is full
`peek`	Returns the head element	Returns `null` if the queue is empty
`poll`	Removes and returns the head element	Returns `null` if the queue is empty
`put`	Adds an element	Blocks if the queue is full
`remove`	Removes and returns the head element	Throws a `NoSuchElementException` if the queue is empty
`take`	Removes and returns the head element	Blocks if the queue is empty

Blocking Queue Example

```
1  package blockingQueue;
2
3  import java.io.*;
4  import java.nio.charset.*;
5  import java.nio.file.*;
6  import java.util.*;
7  import java.util.concurrent.*;
8  import java.util.stream.*;
9
10 /**
11  * @version 1.03 2018-03-17
12  * @author Cay Horstmann
13  */
14 public class BlockingQueueTest
15 {
16    private static final int FILE_QUEUE_SIZE = 10;
17    private static final int SEARCH_THREADS = 100;
18    private static final Path DUMMY = Path.of("");
19    private static BlockingQueue<Path> queue = new ArrayBlockingQueue<>(FILE_QUEUE_SIZE);
20
21    public static void main(String[] args)
22    {
```

```
       try (var in = new Scanner(System.in))
       {
          System.out.print("Enter base directory (e.g. /opt/jdk-11-src): ");
          String directory = in.nextLine();
          System.out.print("Enter keyword (e.g. volatile): ");
          String keyword = in.nextLine();

         Runnable enumerator = () -> {
            try
            {
               enumerate(Path.of(directory));
               queue.put(DUMMY);
            }
            catch (IOException e)
            {
               e.printStackTrace();
            }
            catch (InterruptedException e)
            {
            }
         };

         new Thread(enumerator).start();
         for (int i = 1; i <= SEARCH_THREADS; i++) {
            Runnable searcher = () -> {
               try
               {
                  var done = false;
                  while (!done)
                  {
                     Path file = queue.take();
                     if (file == DUMMY)
                     {
                        queue.put(file);
                        done = true;
                     }
                     else search(file, keyword);
                  }
               }
               catch (IOException e)
               {
                  e.printStackTrace();
               }
               catch (InterruptedException e)
               {
               }
            };
```

Blocking Queue Example

```
            new Thread(searcher).start();
         }
      }
   }

   /**
    * Recursively enumerates all files in a given directory and its subdirectories.
    * See Chapters 1 and 2 of Volume II for the stream and file operations.
    * @param directory the directory in which to start
    */
   public static void enumerate(Path directory) throws IOException, InterruptedException
   {
      try (Stream<Path> children = Files.list(directory))
      {
         for (Path child : children.collect(Collectors.toList()))
         {
            if (Files.isDirectory(child))
               enumerate(child);
            else
               queue.put(child);
         }
      }
   }

   /**
    * Searches a file for a given keyword and prints all matching lines.
    * @param file the file to search
    * @param keyword the keyword to search for
    */
   public static void search(Path file, String keyword) throws IOException
   {
      try (var in = new Scanner(file, StandardCharsets.UTF_8))
      {
         int lineNumber = 0;
         while (in.hasNextLine())
         {
            lineNumber++;
            String line = in.nextLine();
            if (line.contains(keyword))
               System.out.printf("%s:%d:%s%n", file, lineNumber, line);
         }
      }
   }
}
```

Blocking Queue Example - java.util.concurrent

java.util.concurrent.ConcurrentLinkedQueue<E> 5

- `ConcurrentLinkedQueue<E>()`

constructs an unbounded, nonblocking queue that can be safely accessed multiple threads.

java.util.concurrent.ConcurrentSkipListSet<E> 6

- `ConcurrentSkipListSet<E>()`
- `ConcurrentSkipListSet<E>(Comparator<? super E> comp)`

constructs a sorted set that can be safely accessed by multiple threads. Th constructor requires that the elements implement the `Comparable` interface.

java.util.concurrent.ConcurrentHashMap<K, V> 5
java.util.concurrent.ConcurrentSkipListMap<K, V> 6

- `ConcurrentHashMap<K, V>()`
- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`

constructs a hash map that can be safely accessed by multiple threads. The default for the initial capacity is `16`. If the average load per bucket exceeds the load factor, the table is resized. The default is `0.75`. The concurrency level is the estimated number of concurrent writer threads.

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

constructs a sorted map that can be safely accessed by multiple threads. The first constructor requires that the keys implement the `Comparable` interface.

- *Thread-safe collections (implementations of maps and queues).*

Atomic Update of Map Entries

- *Thread unsafe way:*

```
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // ERROR--might not replace oldValue
```

Safer approach:

```
var map = new ConcurrentHashMap<String,
AtomicLong>;
String word = …
…
AtomicLong oldvalue;
...
map.replace(word,oldvalue,new
AtomicLong(newvalue));
```

Bulk Operations on Concurrent Hash Maps.

Three kinds of operations:
- search applies a function to each key/value pairs until a function yields a non-null result and then the search terminates, the search result is returned.
- reduce combines all key/value pairs using provided accumulation function.
- forEach applis a function to all key/value pairs.

Operations:
- operationKeys, operationValues, operation, operationEntries.

```
searchKeys(long threshold, BiFunction<? super K, ? extends U> f)
searchValues(long threshold, BiFunction<? super V, ? extends U> f)
search(long threshold, BiFunction<? super K, ? super V,? extends U> f)
searchEntries(long threshold, BiFunction<Map.Entry<K, V>, ? extends U> f)
```

Bulk Operations on Concurrent Hash Maps.

Some examples.

```
String result = map.search(threshold, (k, v) -> v > 1000 ? k : null);
```

For each map entry print key/value pair

```
map.forEach(threshold,
   (k, v) -> System.out.println(k + " -> " + v));
```

Transformer → consumer variant. Pass output of transformer to consumer.

```
map.forEach(threshold,
   (k, v) -> k + " -> " + v, // transformer
   System.out::println); // consumer
```

Bulk Operations on Concurrent Hash Maps.

Some examples.

reduce operations on key/value pairs.

```
Long sum = map.reduceValues(threshold, Long::sum);
```

reduce operations transformer → accumulator variant.

```
Integer maxlength = map.reduceKeys(threshold,
   String::length, // transformer
   Integer::max); // accumulator
```

Parallel Array Algorithms

Arrays class has number of parallelized operations, e.g. Arrays.parallelSort(), Arrays.parallelSetAll() etc.

```
var contents = new String(Files.readAllBytes(
   Path.of("alice.txt")), StandardCharsets.UTF_8); // read file into string
String[] words = contents.split("[\\P{L}]+"); // split along nonletters
Arrays.parallelSort(words);
```

Sort with a comparator method.

```
Arrays.parallelSort(words, Comparator.comparing(String::length));
```

Compute an operator on all array values.

```
Arrays.parallelSetAll(values, i -> i % 10);
   // fills values with 0 1 2 3 4 5 6 7 8 9 0 1 2 . . .
```

Conditional Variables/Objects

Thread may enter CS but not proceed unless a certain condition is met.

```
public void transfer(int from, int to, int amount)
{
   bankLock.lock();
    try
    {
       while (accounts[from] < amount)
       {
          // wait
          . . .
       }
       // transfer funds
       . . .
    }
    finally
    {
       bankLock.unlock();
    }
 }
```

Consider: One thread enters CS but cannot proceed unless there are enough funds. However, others can't add money to the bank either as others can't acquire the bankLock now.

Conditional Variables/Objects

```
class Bank
{
   private Condition sufficientFunds;
   . . .
   public Bank()
   {
      . . .
      sufficientFunds = bankLock.newCondition();
   }
}
```

```
public void transfer(int from, int to, int amount)
{
   bankLock.lock();
   try
   {
      while (accounts[from] < amount)
         sufficientFunds.await();
      // transfer funds      . . .
      sufficientFunds.signalAll();
   }
   finally
   {
      bankLock.unlock();
   }
}
```

Current thread gives up lock and allows others to enter CS. Goes into `wait' state until another thread calls the signalAll() method. Problems: Could lead to deadlock if no one else calls signalAll().

```
 1  package synch;
 2
 3  import java.util.*;
 4  import java.util.concurrent.locks.*;
 5
 6  /**
 7   * A bank with a number of bank accounts that uses locks for
serializing access.
 8   */
 9  public class Bank10  {
11     private final double[] accounts;
12     private Lock bankLock;
13     private Condition sufficientFunds;
14
15     /**
16      * Constructs the bank.
17      * @param n the number of accounts
18      * @param initialBalance the initial balance for each
account
19      */
20     public Bank(int n, double initialBalance)
21     {
22        accounts = new double[n];
23        Arrays.fill(accounts, initialBalance);
24        bankLock = new ReentrantLock();
25        sufficientFunds = bankLock.newCondition();
26     }
27
28     /**
29      * Transfers money from one account to another.
30      * @param from the account to transfer from
31      * @param to the account to transfer to
32      * @param amount the amount to transfer
33      */
34     public void transfer(int from, int to, double amount)
throws InterruptedException
35     {
36        bankLock.lock();
37        try
38        {
39           while (accounts[from] < amount)
40              sufficientFunds.await();
41           System.out.print(Thread.currentThread());
42           accounts[from] -= amount;
43           System.out.printf(" %10.2f from %d to %d", amount,
from, to);
44           accounts[to] += amount;
45           System.out.printf(" Total Balance: %10.2f%n",
getTotalBalance());
46           sufficientFunds.signalAll();
47        }
48        finally
49        {

50           bankLock.unlock();
51        }
52     }
53
54     /**
55      * Gets the sum of all account balances.
56      * @return the total balance
57      */58     public double getTotalBalance()
59     {
60        bankLock.lock();
61        try
62        {
63           double sum = 0;
```

```
64
65          for (double a : accounts)
66             sum += a;
67
68          return sum;
69       }
70       finally
71       {
72          bankLock.unlock();
73       }
74    }
75
76    /**
77     * Gets the number of accounts in the bank.
78     * @return the number of accounts
79     */
80    public int size()
81    {
82       return accounts.length;
83    }
84 }
```

java.util.concurrent.locks.Lock 5

- `Condition newCondition()`

  returns a condition object associated with this lock.

java.util.concurrent.locks.Condition 5

- `void await()`

  puts this thread on the wait set for this condition.

- `void signalAll()`

  unblocks all threads in the wait set for this condition.

- `void signal()`

  unblocks one randomly selected thread in the wait set for this condition.

Synchronized keyword and Intrinsic Locks

Every object has an intrinsic lock.

Synchronized ← Keyword that uses the intrinsic lock.

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

```
public void method()
{
    this.intrinsicLock.lock();
    try
    {
        method body
    }
    finally
    {
        this.intrinsicLock.unlock();
    }
}
```

Synchronized keyword and Intrinsic Locks

`wait()` ***and*** `notify()/notifyAll()`

`intrinsicCondition``.await()` ***and***
`intrinsicCondition``.signalAll()`