320341 Programming in Java

JACOBS
UNIVERSITY

Fall Semester 2014

Lecture 11: Introduction to Concurrency

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Introduction

JACOBS UNIVERSITY

Multithreaded Programs

- A program represents separate, independently running subtasks

- Threads allow multiple activities to proceed concurrently

  - ❑ Example: typing data & printing at the same time in a text editor

  - ❑ Each independent task is called a **Thread**

- *A single program can have multiple concurrently executing threads*

Introduction

JACOBS UNIVERSITY

Multithreaded Programs

- Programs that run more than one thread at the same time

When to use Multithreading?

- When part of your program is tied to a particular event or resource, you don’t want to hang-up the whole program

- Create a **Thread** to handle that part of the program

Definitions

JACOBS UNIVERSITY

Multitasking

- A multitasking operating system can run more than one process at a time
- CPU time is periodically provided to each process

Preemptive Multitasking

- OS interrupts programs without consulting with them to release the CPU
- E.g., Unix/ Linux, Windows NT/ XP, Windows 9x (32-bit) and OS X

Co-operative (non-preemptive) Multitasking

- Programs only interrupted when they are willing to yield control
- E.g., Windows 3.x, Mac OS 9, OSs on devices like cell phones

Running Tasks in Separate Threads

JACOBS UNIVERSITY

Two approaches for implementing threads

1. Implement **Runnable** interface (**Thread** class separate from main class)

2. Inherit from **Thread** class (Makes the main class a **Thread**)
   - This approach is no longer recommended!

Running Tasks in Separate Threads

JACOBS UNIVERSITY

Approach 1: Implement **Runnable** interface (recommended approach)

1. Place the code for the task into the *run* method of a class that implements the **Runnable** interface

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

Example

```
class MyRunnable implements Runnable {
     public void run() {
          // task code
     }
}
```

Running Tasks in Separate Threads

JACOBS UNIVERSITY

Approach 1: Implement **Runnable** interface (recommended approach continued …)

2. Construct object of your class

```
Runnable r = new MyRunnable();
```

3. Construct a **Thread** object from the **Runnable**

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

4. Start the **Thread** using the *start* method

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

Tasks inside the *run* method can now be executed in parallel!

Running Tasks in Separate Threads

Example

Independent **Thread**

```
class BallRunnable implements Runnable {
        …
        public void run()
                    try {
                        for (int i = 1; i <= STEPS; i++) {
                                ball.move(component.getBounds());
                                component.repaint();
                               Thread.sleep(DELAY);
                          }
                                                                    }
catch(InterruptedException e) {}                                    }

…
}
```

- Running the **Thread** (e.g. inside main class)

```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable(b, panel);
Thread t = new Thread(r);
t.start();
```

Running Tasks in Separate Threads

JACOBS UNIVERSITY

Approach 2: Inherit from **Thread** class (Not Recommended!)

- Define Threads using a subclass of the **Thread** class
- The **Thread** class has all methods necessary to create and run threads

Thread class implements **Runnable** interface

```
class MyThread extends Thread {
        …
        public void run()   {
                try {
                        for (int i = 1; i <= STEPS; i++) {
                                ball.move(component.getBounds());
                                component.repaint();
                                Thread.sleep(DELAY);
                                                }                       }
catch(InterruptedException e) {}
                                        }
                                                …
                                }
```

Override the *run* method to make thread handle task

Code to be executed “simultaneously” with the other threads in the program

Running Tasks in Separate Threads

JACOBS UNIVERSITY

Example

```
public class SimpleThread extends Thread {
    private int countDown = 5;    private static int threadCount = 0;
    private int threadNumber = ++threadCount;
    public SimpleThread () {  System.out.println("Making " + threadNumber);
  }
  public void run () {
     while(true) {
        System.out.println ("Thread " + threadNumber + "(" + countDown + ")");
        if(--countDown == 0) return;
     }
  }
  public static void main (String[] args) {
     for (int i = 0; i < 5; i++)
        new SimpleThread ().start();
     System.out.println ("All Threads Started");
  }
}
```

Thread loop

Thread creation

Condition to break out of loop

Running Tasks in Separate Threads

JACOBS UNIVERSITY

Sample Output

```
Making 1
Making 2
Thread 1(5)
Making 3
Thread 1(4)
Thread 2(5)
Thread 3(5)
Thread 1(3)
Thread 2(4)
Making 4
Thread 3(4)
Thread 1(2)
Thread 2(3)
Thread 3(3)
Making 5
Thread 4(5)
Thread 1(1)
Thread 2(2)
Thread 3(2)
All Threads Started
Thread 5(5)
Thread 4(4)
Thread 2(1)
Thread 3(1)
Thread 5(4)
Thread 4(3)
Thread 5(3)
Thread 4(2)
Thread 5(2)
Thread 4(1)
Thread 5(1)
```

Note:

- Each thread gets a portion of the CPU time to execute
- Threads are not run in the order in which they were created!

Daemon Threads

JACOBS UNIVERSITY

Provides general services in the background while the program is running

- The program terminates when all non-daemon threads have completed
- Daemon threads are not part of the essence of the program

How to find if a thread is daemon

- Call the method ***isDaemon( )*** of the **Thread** class
- If a thread is a daemon, all threads it creates are daemon

Setting a thread to daemon

- Call the **Thread** method *setDaemon()*

Daemon Threads

JACOBS UNIVERSITY

Example

Sets thread to daemon

Starts thread

new thread created

```
class Daemon extends Thread {
      ...
      public Daemon() {
      setDaemon(true);
      start();
      }
      public void run() {
            while (true)
             yield();
      }
}

public class Daemons {
      public static void main (String[] args) throws IOException {
      Thread d = new Daemon();
            System.out.println("d.isDaemon() = " + d.isDaemon());
      }
}
```

Interrupting Threads

JACOBS UNIVERSITY

The interrupt method can be used to request termination of a thread

- Each thread should occasionally check if it is interrupted
- First call **Thread**.*currentThread* to get current thread
- Next call is *interrupted* method

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

- If the interrupted method is called on a blocked thread, the blocking call (such as *sleep* or *wait*) is terminated by an **InterruptedException**

Interrupting Threads

The run method has the form

```
public void run()
      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

}
```

Example

The run method containing *sleep* method

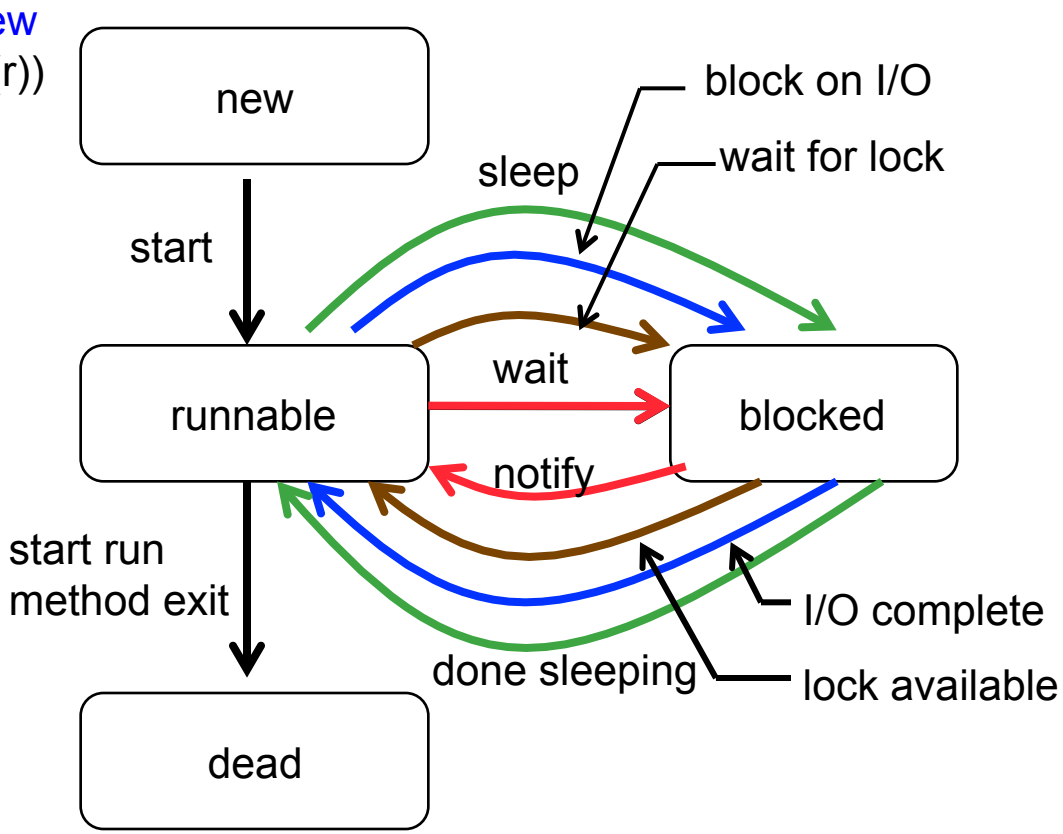
```
public void run()
        try {
            ....
            while (more work to do) {
                  do more work
                  Thread.sleep(DELAY);
            }
        } catch(InterruptedException e) {
              // thread was interrupted during sleep or wait
                                  } finally {
                                        cleanup, if required
                                                          }
                                                        // exiting the run method
terminates the thread                                         }
```

The *sleep* method calls **InterruptedException** if thread is interrupted, so no need to check for thread interruption

Thread States

Threads can be in one of four states (New, Runnable, Blocked or Dead)

Thread created with new operator (new **Thread**(r))



Preemptive scheduling

- Thread is given a time-slice

Cooperative scheduling

- Thread loses control when it calls method like *sleep* or *yield*

- A blocked thread reenters runnable state using the same route that blocked it!
- Online simulation: http://courses.cs.vt.edu/~csonline/OS/Lessons/Processes/index.html

Dead Threads

A thread is dead for two reasons

1. Dies a natural death when the *run* method exits normally
2. Dies abruptly because an uncaught exception terminates the run

Find out if a thread is alive (in runnable or blocked state)

- Call *isAlive* method – returns true if thread is in runnable or blocked state

Thread Properties

Thread Priorities

- Every **Thread** has a priority
- By default, a **Thread** inherits the priority of its parent thread
- Increase priority of a thread by using *setPriority* method

- Priority values (defined in **Thread** class)
  - MIN_PRIORITY
  - MAX_PRIORITY
  - NORM_PRIORITY

- Thread priorities are system dependent (e.g., in Sun JVM for Linux, priorities are ignored)

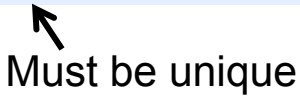
Thread Groups

Thread Groups make it possible to work with a group of threads

- Threads are categorized according to functionality

Constructing a **Thread** Group

```
String groupName = . . .;
ThreadGroup g = new ThreadGroup (groupName)
```



Add threads to the Group

```
Thread t = new Thread (g, threadName)
```

Thread Groups

JACOBS UNIVERSITY

Find out if any threads of a particular group are still runnable

```
if (g.activeCount() == 0)  {
   // all threads in the group g have stopped
}
```

Interrupt all threads in a group

```
g.interrupt(); // interrupts all threads in group g
```

JACOBS UNIVERSITY

Race condition

- When two or more threads share access to the same object and if each calls a method that modifies the state of the object, corrupted objects can result

Synchronization Approaches in Java

- **Exclusion synchronization**
- **Condition synchronization**

Synchronization

JACOBS UNIVERSITY

Exclusion synchronization

- Synchronization by *mutual exclusion*

Condition synchronization

- Synchronization by *cooperating threads*
- Makes use of exclusion synchronization

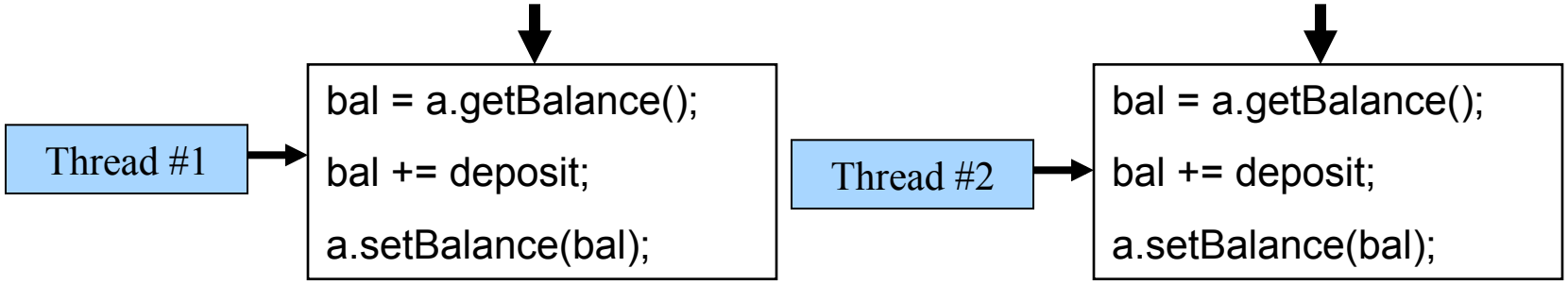
Synchronization

Example: Suppose you want to perform the steps below using threads

- Fetch bank balance
- Increase balance by deposit amount
- Write results back to account record

get-modify-set sequence on shared Resources must be synchronized

Synchronization



Race Condition

- Thread #1 and thread #2 can modify *bal* in an interleaved way

- How to prevent 2 threads from simultaneously writing & reading same object?

Synchronization

Synchronization is required for reliable communication between threads as well as for mutual exclusion

Java uses 2 mechanisms to protect code block from concurrent access

- Use **synchronized** keyword OR
- **ReentrantLock** (JDK 5.0)

ReentrantLock

JACOBS UNIVERSITY

Basic outline for protecting code block

```
myLock.lock(); // a ReentrantLock object
try  {
        critical section
} finally {
        myLock.unlock(); // make sure the lock is unlocked even if an exception is thrown
}
```

Guarantees that only one thread can enter the critical section (Exclusive synchronization)

Other threads are blocked until the 1st thread unlocks the lock object

- The lock is called *reentrant* because a thread can repeatedly acquire a lock it already owns

- The thread has to call *unlock* for every call to *lock*, in order to relinquish lock

ReentrantLock

JACOBS UNIVERSITY

Example: Money Transfer in a Bank

```
public class Bank {
   private Lock bankLock = new ReentrantLock();
    . . .
   public void transfer(int from, int to, int amount)  {
     bankLock.lock();
     try  {
        if (accounts[from] < amount) return;
        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();
     }
   }

}
```

Serialized access

ReentrantLock implements **Lock** interface

ReentrantLock

JACOBS UNIVERSITY

Which Code Blocks to Protect?

- Code blocks that require multiple operations to update or inspect a data structure

Lock class package

- java.util.concurrent.locks

Condition Objects

Use **Condition objects** to manage threads that have acquired a lock but cannot do useful work

- Condition objects are also called **Condition variables**

If condition not met, thread must wait

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

- While the thread is waiting for its condition to be fulfilled, no other thread can access the lock

Condition Objects

A lock object can have one or more associated condition objects

- Obtain a condition object with the **newCondition** method

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

- If the transfer method finds that the sufficient funds are not available, it calls

  `sufficientFunds.await()`

- The current thread is deactivated and gives up the lock
  - ❑ Enters the **wait set** for that condition
  - ❑ Remains deactivated until another thread calls **`signalAll`** on the same condition

Condition Objects Example

Call **signalAll()** after finishing transferring the money

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

         sufficientFunds.signallAll();
       }
    }  finally {
      bankLock.unlock();
   }
}
```

- Call to **signalAll()** does not immediately activate a waiting thread
- It only makes it available to compete

Reentrant/ Condition Objects Summary

JACOBS UNIVERSITY

A **Lock** protects sections of code

- Allows only one **Thread** to execute the code at a time

A **Lock** manages threads that enter a protected segment

A **Lock** can have one or more associated **Condition** objects

Each **Condition** object manages threads that have entered a protected code section but that cannot proceed

The synchronized Keyword

Used to implement **exclusive synchronization** prior to JDK 5.0

All Java objects have potential for exclusion synchronization

- Every Java object has a **monitor** and a **monitor lock** (or **intrinsic lock**)

- The **monitor** ensures that its object's **monitor lock** is held by a max of only one thread at any time

- **Monitors** and **monitor locks** are used to enforce **mutual exclusion**

Threads are synchronized by locking objects before accessing critical section

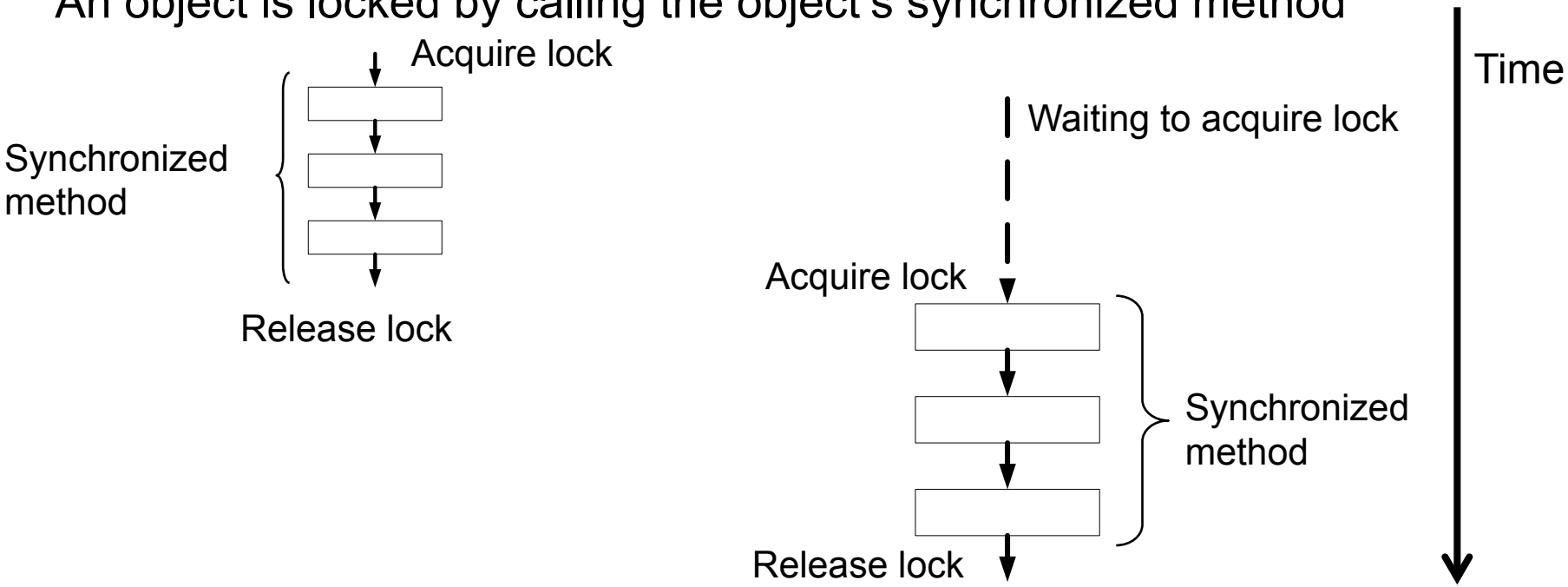
The synchronized Keyword

Java implements synchronization with synchronized keyword in two ways

- Through synchronized methods
- Through synchronized statement

The synchronized Keyword

An object is locked by calling the object‘s synchronized method



Another **Thread** invoking a synchronized method on the same object must wait until the lock is released

Exclusion synchronization forces execution of two threads to be mutually exclusive in time

The synchronized Keyword

JACOBS UNIVERSITY

Example

```
class Bank  {
   public synchronized void transfer(int from, int to, int amount)
                                          throws InterruptedException {
      while (accounts[from] < amount)
            wait(); // wait on object lock's single condition
      accounts[from] -= amount;
      accounts[to] += amount;
      notifyAll(); // notify all threads waiting on the condition
   }
   public synchronized double getTotalBalance() { . . . }
   private double accounts[];
}
```

- The implicit object lock has a single associated condition
- The ***wait*** method adds a Thread to the ***wait set***
- ***notifyAll*** / ***notify*** methods unblock waiting threads

Lock / Condition or Synchronized methods?

JACOBS UNIVERSITY

- Prefer the **synchronized** keyword – write less code
- Use **Lock/ Condition** if you specifically need the additional power

Static Synchronized Methods

JACOBS UNIVERSITY

Static methods can be declared **synchronized**

- Acquire a lock on the **Class** object for the class
- Two threads cannot execute *static synchronized methods* of the same class at the same time

If static data is shared between threads, then access to it must be protected using *static synchronized methods*

No effect on objects of the class

Only other static synchronized methods are blocked

Synchronized Statements

Allows to execute synchronized code that acquires the lock of any object

Allows to synchronously execute block of code within a method

A synchronized statement has two parts:

- An object whose lock is to be acquired
- A statement block to execute when the lock is obtained

Synchronized Statements

JACOBS UNIVERSITY

General form

```
synchronized (expression) {
        statement block
}
```

- *expression* must evaluate to an object reference
- Statement block to execute when lock on referenced object is obtained

Synchronized Statements

JACOBS UNIVERSITY

Example

```
/* make all elements in the array non-negative */
public static void abs (int [] values) {
        synchronized (values) {
            for (int i=0; i< values.length; i++)
{                                                                    if
(values[i]<0)
                    values[i] = -1 * values[i];
                }
        }
}
```

Exclusive access to code block

- The values array contains elements to be modified and has been synchronized
- Guaranteed that the loop can execute without values being modified
- This kind of synchronization sometimes called **client-side synchronization**

Nested Critical Sections

JACOBS UNIVERSITY

Critical sections can be nested

```
class V ... {
      synchronized void m( ) {
            synchronized (wo )  {
                        statement block B
            }
      }
}
```

- Block **B** executes with exclusive access to both `wo` and current instance of `V`
- Call a synchronized method inside another synchronized method

- Useful for coordinating updates e.g., between say objects `z`, `x` and `y` given:
  - A method **`m`** on **`z`** must also update **`x`** and `y` to maintain consistency
  - **`m`** can contain nested blocks synchronized with respect to `x` and `y`
  - Innermost block provides exclusive access to `x`, `y` and `z`

Multiple Locks on Object

We can nest methods and blocks that are synchronized with respect to the same object

Assume:

- Thread `t` calls `o.m` and that `m` is synchronized
- Let `t` call another synchronized method on `o` from within `m; t` gets an additional lock on `o`
- Each time `t` exits a critical section, its releases a lock
- Thus `t` keeps `o` locked until it exits its outermost critical section wrt `o`
- Excessive multiple locks can result in performance degradation

Volatile Fields

The volatile keyword offers a lock-free mechanism for synchronizing access to an instance field

If a field is declared as volatile, the compiler and the virtual machine take into account that the field may be concurrently updated by another thread

```
private volatile boolean done;
```

Forces the VM not to cache the instance field

Accessing volatile variables is slower than accessing regular variables

Deadlocks

Multi-threading and synchronization create the danger of **deadlock**

Deadlock: *a circular dependency on a pair of synchronized objects*

Suppose that:

- One thread locks object **X**
- Another thread locks object **Y**
- The first thread tries to call a synchronized method on object **Y**
- The second thread tries to call a synchronized method on object **X**

The result: *the threads wait forever – deadlock*

Deadlocks

JACOBS UNIVERSITY

Example

- Bank Application - Money transfer

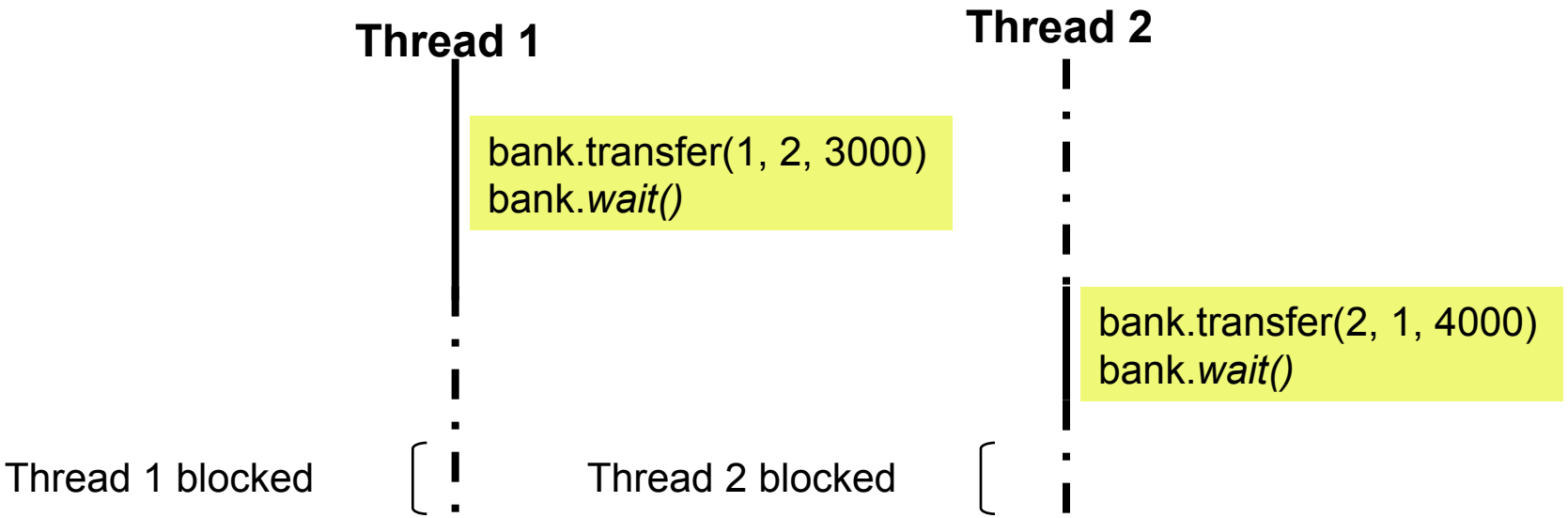
```
class Bank  {
   public synchronized void transfer(int from, int to, int amount) throws
                                          InterruptedException {
      while (accounts[from] < amount)
         wait(); // wait on object lock's single condition
      accounts[from] -= amount;
      accounts[to] += amount;
      notifyAll(); // notify all threads waiting on the condition
   }
   public synchronized double getTotalBalance() { . . . }
   private double accounts[];
}
```

A transfer is possible if sufficient funds are available, otherwise the thread must wait

Deadlocks

Example cont …

- Account 1: $2,000
- Account 2: $3,000
- Thread 1: Transfer $3,000 from Account 1 to Account 2
- Thread 2: Transfer $4,000 from Account 2 to Account 1



- Thread 1 & Thread 2 are deadlocked!

Deadlocks

There is nothing in the Java programming language to avoid or break deadlocks

Design your program to ensure that a deadlock situation cannot occur

Fairness

JACOBS UNIVERSITY

Fair Locking Policy

- Favors a Thread that has been waiting the longest

How to specify a fair locking policy?

```
Lock fairLock = new ReentrantLock (true);
```

A fair locking policy can cause a drag on performance

Also, no guarantee that the thread scheduler will be fair

Lock Testing

A Thread blocks indefinitely when it calls lock method when the lock is owned by another **Thread**

Use the `tryLock` method to try and acquire the lock when its available, and do something if its not

```
if (myLock.tryLock())
      // now the thread owns the lock
     try  {
          ...
     }  finally  {
         myLock.unlock();
     }
else  // do something else
```

- *If the lock is available when the call is made, the current thread gets it, even if another thread has been waiting to lock it*

Read/ Write Locks

JACOBS UNIVERSITY

Two lock classes in `java.util.concurrent.locks` package

- **ReentrantLock**
- **ReentrantReadWriteLock**

The **ReentrantReadWriteLock** class is useful when more reads than writes occur in a data structure

- Allow shared access for readers
- Writers must still have exclusive access

Read/ Write Locks

JACOBS UNIVERSITY

Steps for using **ReentrantReadWriteLock** object

1. Construct a **ReentrantReadWriteLock** object

```
private ReentrantReadWriteLock rwl = new
                                    ReentrantReadWriteLock();
```

2. Extract Read and Write locks

```
private Lock readLock = rwl.readLock();
private Lock writeLock = rwl.writeLock();
```

Read/ Write Locks

JACOBS UNIVERSITY

3. Use the read lock in all accessors

```
public double getTotalBalance() {
  readLock.lock();
  try {. . .}
  finally { readLock.unlock(); }
}
```

4. Use write locks in all mutators

```
public void transfer(. . .) {
  writeLock.lock();
  try {. . .}
  finally { writeLock.unlock(); }
}
```

Producer/ Consumer Problem

JACOBS UNIVERSITY

The producer/ consumer problem is characterized as follows:

- Producer generates data and stores it in a shared buffer
- The consumer read data from shared buffer

The following property must be preserved

- If the buffer is not full, the producer may continue to produce, otherwise it must wait for the consumer to “consume” to create space in the buffer
- If the buffer is not empty, the consumer may continue to consume, otherwise it must wait for the producer to “produce” and add to the buffer

Blocking queues are useful for *coordinating multiple threads*

- *Producer threads/ consumer threads*
- *http://courses.cs.vt.edu/~csonline/OS/Lessons/Synchronization/index.html*

Blocking Queues

JACOBS UNIVERSITY

A Queue has two fundamental operations

- Adds a data element to the tail of the queue
- Removes a data element from the head of the queue

A queue is blocking if

- It causes a Thread to block if an *add* operation is invoked when the queue is full
- It causes a Thread to block when a *remove* operation is invoked when queue is empty

Blocking Queues

Example: Producer Thread

puts elements into queue

Recursively puts elements into queue

```
class FileEnumerationTask implements Runnable {
    ...
    public void run() {
      try {
          enumerate(startingDirectory);
          queue.put(DUMMY);
      } catch (InterruptedException e) {}
    }

    public void enumerate(File directory) throws InterruptedException {
        File[] files = directory.listFiles();
        for (File file : files)     {
            if (file.isDirectory()) enumerate(file);
            else queue.put(file);
        }
    }
    public static File DUMMY = new File("");
    private BlockingQueue<File> queue;
    private File startingDirectory;
}
```

Blocking Queues

JACOBS UNIVERSITY

Example: Consumer Thread

removes elements from the queue

```
class SearchTask implements Runnable   {
    …
    public void run() {
        try {
          boolean done = false;
          while (!done) {
            File file = queue.take();
            if (file == FileEnumerationTask.DUMMY) { queue.put(file); done = true; }
            else search(file);
          }
        }
        catch (IOException e) { e.printStackTrace(); }
        catch (InterruptedException e) {}
    }
    public void search(File file) throws IOException   {
          …
     }
    private BlockingQueue<File> queue;
    private String keyword;
  }
```

Thread-Safe Collections

JACOBS UNIVERSITY

Use thread-safe collections for multi-threaded usage

- **ConcurrentHashMap**
- **ConcurrentLinkedQueue**

ConcurrentHashMap

- Efficiently supports large number of readers & fixed number of writers
- Default – 16 simultaneous writers
- Uses sophisticated algorithms that never locks entire table
- Atomic insertion and removal operations

```
cache.putIfAbsent(key, value);
```

```
cache.remove(key, value);
```

```
cache.replace(key, oldValue, newValue);
```

Older Thread-Safe Collections

JACOBS UNIVERSITY

Vector and **Hashtable** classes provided thread-safe implementations of a dynamic array and a hash table

- **Vector** now replaced by **ArrayList** (not thread safe)
- **Hashtable** now replaced by **HashMap** (not thread safe)

Make a collection thread-safe by using synchronization wrapper

```
List synchArrayList = Collections.synchronizedList(new ArrayList());
Map synchHashMap = Collections.synchronizedMap(new HashMap());
```

Must use synchronized block to iterate over the collection

```
synchronized (synchHashMap) {
        Iterator iter = synchHashMap.keySet().iterator();
        while (iter.hasNext()) . . .;
}
```

Callable and Futures

JACOBS UNIVERSITY

Callable interface

- Encapsulates a task that runs asynchronously, but returns a value

- The **Callable** interface is similar to the **Runnable** interface, save for returning a value

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Callable and Futures

Future interface

- Represents the results of an *asynchronous computation*
- Provides methods
  - To check if the computation is complete (`isDone`)
  - To check if the computation is complete (*using timeout*)
  - To retrieve the result of computation (`get`)

```
public interface Future<V>  {
    V get() throws . . .;
    V get(long timeout, TimeUnit unit) throws . . .;
    void cancel(boolean mayInterrupt);
    boolean isCancelled();
    boolean isDone();
}
```

Blocks until the computation is complete

Callable and Futures

FutureTask

- A wrapper class that implements both the **Future** and **Runnable** interfaces

Callable and Futures: Example

```
class MatchCounter implements Callable<Integer> {
        public Integer call()  {
        try  {
            File[] files = directory.listFiles();
            ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
            for (File file : files)
                if (file.isDirectory())  {
                    MatchCounter counter = new MatchCounter(file, keyword);
                    FutureTask<Integer> task = new FutureTask<Integer>(counter);
                    results.add(task);
                    Thread t = new Thread(task);
                     t.start();
                } else  {  if (search(file)) count++;   }
             for (Future<Integer> result : results)
                    try {   count += result.get();
                    }  catch (ExecutionException e)  {
                            e.printStackTrace();
                    }
        }  catch (InterruptedException e) {}
        return count;
    } …}
```

Similar to implements **Runnable**

Array to store task results

Future task object to store tasks

Results are read only through the *get* method;
Each call to *get* blocks until result is available

Guidelines

If an action takes a long time, fire up a new thread to do the work

If an action can block on I/O, fire up a new thread to do the work

If you need to wait for a specific amount of time, don't sleep in the event dispatch a thread - use timer events

Threads and Swing

Swing is not thread safe

- Avoid manipulating IU elements from multiple threads

Work done in threads cannot touch the UI

Read any information from the UI before launching threads

Launch them and then update the UI from the event dispatching thread once the threads have completed

- This is often called **“single thread rule for Swing programming”**

Reading Assignment

JACOBS UNIVERSITY

- Horstmann, C. S. & Cornell, G. (2013) Core Java(TM) 2 (Vol. I), Prentice Hall, 9th Edition. Chapter 14.
- Deitel, P. J. & Deitel, H. M (2012) Java How to Program. 9th Ed. Pearson Education International. Chapter 23.

Practice Task

- There is a one-way bridge that can hold up to three cars. Cars arrive at one end of the bridge and exit the bridge at the other end. Traffic is allowed only in the one, available, direction. Describe a solution to this synchronization problem. Write a Java program to solve the problem.