Assignment 4

- Due Tuesday July 27th, 8 PM
- **No late assignments accepted**
- You will have a chance to demo your GUI during your lab on the 28th
- Sign up for a demo time-slot during your lab today
- If working with a partner, you do **one** demo together
- People within a given lab section have first dibs on the time-slots in that section
- After the labs today, I will post a time-sheet indicating remaining open slots

Review

Recursive Methods

- We have seen that a method can make a call to another method (e.g. a method calling a helper method).
- Many programming languages, including Java, allow a method to make a call to itself – we call this *recursion*.
- A method that makes a call to itself is known as a *recursive method*.
- When a method calls itself, it is essentially repeating itself and so recursion is a form of looping.
- Note that in some programming languages, recursion is the *only* way to loop through a block of code.

Recursive Methods

- Some problems are more naturally solved using recursion than a looping construct such as a *for* loop.
- Problems whose solution can be defined in terms of solutions to *smaller* sub-problems have natural recursive solutions.
- There are also some data structures whose *structure* can be defined recursively (a binary tree, for example). These structures can be processed recursively in a very natural way.
- We'll start with some easy examples.

Real-World Examples

- Shampoo bottle instructions
  - Lather
  - Rinse
  - **Repeat**

repeat all three steps, *including the repeat step*

- An unhelpful dictionary definition
  - Book (n.) - A bunch of pages that make up a **book**
- Neither of these ever terminate – they keep calling themselves

Terminating Conditions

- We need a defined stopping point
  - e.g. "If hair is clean, stop. Otherwise, repeat."
- Without this, you get infinite recursion, and eventually a memory overload error

Recursive Method Calls – General Form

- Our `drawRamp` method illustrates the general form of a recursive method call:

```
type recursiveMethod( type param1, type param2,… )
{
    if( base case )
       // handle base case (code omitted)
    else
    {
         // operations to do before recursive call
         // (code omitted)
         recursiveMethod( … );  // recursive call
         // operations to do after recursive call
         // (code omitted)
    }
}
```


Recursion vs Iteration (cont'd)

- Recursion usually requires more memory than iteration
  - each method call creates a new stack frame in which its parameters and local variables are stored
- Sometimes recursion is more natural so it may take more time to develop an iterative solution.
- Rule of thumb:
  - use iteration when it is easy and natural to do so.
  - use recursion when it is easy and natural to do so.

Conclusion

- Recursion can add simplicity, elegance and readability to a program
- Not always the most efficient method
- Check whether you could solve the problem more efficiently in an iterative fashion
- Check whether your problem naturally lends itself to being solved by solving a number of subproblems
  - e.g. Tree traversal

In-Class Exercise I

- We know how to write a method to take an ArrayList<String> and print out each item using a for-loop or an iterator
- Write a recursive method that does the same thing
  - What is your base case?
  - How do you get closer to your base case?

Learning Goals Review

- trace code that uses recursion to determine what the code does
- draw a recursion tree corresponding to a recursive method call
- draw a stack trace of code that uses single and multi-branch recursion
- write recursive methods
- replace a recursive implementation of a method with an iterative solution (may need to use a stack to model the run-time stack)

Threads

Reading

2nd Ed: Chapter 23
3rd and 4th Eds: Chapter 20

Other Resources

http://www.ugrad.cs.ubc.ca/~cs219/CourseNotes/Threads/intro.html
http://download.oracle.com/docs/cd/E17409_01/javase/tutorial/essential/concurrency/index.html

Learning Objectives

- describe the multi-threaded programming model including thread scheduler, thread priority, and time slices.
- describe the various states that a Java thread can achieve and the events that lead to transition from one state to another
- define the terms deadlock, race condition and critical section
- identify possible legal traces of a multithreaded program
- identify deadlock and race conditions in a multithreaded program
- write a thread-safe code using Lock and Condition objects
- identify possible legal traces of a Java program that uses synchronization, locks and conditions

Multitasking

- Many programs you use do more than one thing at a time
  - You can write an email while the email program checks for new message
  - You can type in a word processor while it auto-saves your draft
  - A web browser can load a large photo or a YouTube clip while also loading other pages
- Imagine if in each of these cases, the program could do only one thing at a time in sequential order
  - It would be a huge bottleneck

Multitasking

- Up until now, we have created fairly simple programs that do one thing at a time
- If we had more than one task to do, each task was completed before the next was started
- In contrast, we can use *threads* to develop *concurrent* software
- Java was designed from the ground up to support concurrency, so it's fairly easy compared with some other languages

Multitasking

- A typical program (process) spends a lot of time waiting for events to occur:
  - input from user
  - read data from/write data to a disk, etc.
- Most modern operating systems are **multitasking**: the processor switches between many programs, interleaving their instructions:

- The operating system's **scheduler** decides how programs run.

Multitasking

- Notice that the instructions are interleaved, **not** parallel like this:

program A: ----------
program B: ----------

A typical computer only has a single CPU. Even with multiple CPUs, it will usually have more programs running than it has CPUs. So it runs program A for a little then program B for a little bit, and so on. This gives the illusion of programs running simultaneously.

Concurrent Programming

- Sometimes we need a *single* program to do more than one thing at the same time.
- One way is to create multiple processes, but the overhead from context switching is large for regular processes
  - need to save/restore large amount of state information
- Solution: use ***threads*** **-** ***lightweight*** processes with small state information
  - Allows a program to be split into multiple threads, supporting ***concurrent programming***
  - One thread runs while the others wait to be scheduled to run or for some other condition to occur.

Java Threads

- The JVM supports threads and concurrent programming.
- If a program has more than one thread, a (JVM) scheduler will determine when each thread gets to run.
- There are two types of schedulers:
  - ***pre-emptive***: each thread is allowed to run for a maximum amount of time (a time slice) before it is suspended and another thread is allowed to run
  - ***non pre-emptive***: once a thread is allowed to run it continues to run until it has completed its task or until it explicitly yields to another thread
- The scheduler in *most* JVMs is pre-emptive.
  - As this is not guaranteed, we must not write code that assumes a pre-emptive scheduler.

Java Threads

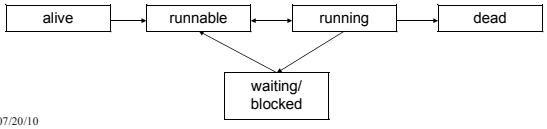
- A thread in Java
  - is an instance of the `Thread` class
  - has a priority and an optional name
  - has a `start()` method
    - this method puts the thread into the *runnable* state so that it can be selected for execution by the thread scheduler
  - has a `run()` method
    - the code in this method is executed when the thread runs
    - it is overridden to specify the particular behaviour of the thread

Thread States

- A thread can be in one of five states:
  - ***alive***
    - the thread has been constructed but start() has not been called
  - ***runnable***
    - the thread is ready to be scheduled to run
  - ***running***
    - the run() method is executing
  - ***waiting/blocking***
    - the thread can't run until some event has occurred (e.g., the passing of a certain amount of time)
  - ***dead***
    - the run() method has run to completion

Thread States

- The diagram below shows the transitions that can occur between these five states.
- For example, a thread can transition from *runnable* to *running,* or vice versa.
- However, a thread cannot transition directly from *runnable* to *dead*.



Thread States

- When a thread dies, its state is still accessible (in other words, the thread object is not destroyed)
- A thread that reaches the dead state cannot be restarted!
  - If you want a thread to run again, just create a new instance of the corresponding class and start it.
- When the JVM starts, it creates a thread that runs `main()`. The JVM continues to execute an application until all user-threads die or `System.exit()` is called.

Creating and Using Threads in Java

Two ways to create a thread

1. Extend *Thread* and override the *run()* method

```
class SumThread extends Thread {
  int  end;
  int sum;

  SumThread( int end ) {
      this.end = end;
  }

  public void run() {
      // sum integers 1, 2,  . . ., end
      // and set the sum
  }
}
```

Creating and Using Threads in Java

To create and start `SumThread` :

```
SumThread  t = new SumThread( 150 );

t.start();
```

Thread t will start running sometime after that

Creating and Using Threads in Java

2. Using the ***Runnable*** interface (has only one method *`run()`*)
   - Define a class that implements ***Runnable***
   - Create an object `obj` of that class
   - Create a thread `t` wrapped around that object
     - `Thread` has a constructor with a Runnable parameter
   2. start `t`
      1. JVM will invoke the `run()` method of `obj`
- Method 2 is preferable when
  - the class that contains `run()` already extends another class
  - you want to separate
    - the code executed by the thread
    - the state info that is maintained by the thread

Runnable interface

- With this second approach, the Thread can be seen as a worker and the Runnable object is a job provided to the worker

Creating and Using Threads in Java

- As mentioned, the second approach is chosen when the class already extends something else (and thus can't extend Thread)
- It's also considered a better design from an OO standpoint
  - We shouldn't subclass Thread unless we are creating a more specific type of Worker and need more specific worker behaviours. If all we need is a new *job* to be carried out by a worker, it's better to implement Runnable and provide that Runnable object (job) to the worker

Creating and Using Threads in Java

- We will see examples of both approaches, however, so that you will be familiar with both

Creating Threads (cont'd)

- Here is the same example in this style:

```
class SumRun implements Runnable {
   int  end;
   int sum;

   SumRun(int  end) {
       this.end = end;
   }

   public void run() {
        // sum integers 1, 2,  . . ., end
        // set sum
   }
 }
```


- To create a thread for SumRun and start it :

```
SumRun  srun = new SumRun(150);

Thread sumRunThread = new Thread(srun);

sumRunThread.start();
```


Runnable Example

```
public class MyRunnable implements Runnable {

   public void run()

   {

      go();

   }

   public void go()

   {

      doMore();

   }


   public void doMore()

   {

      System.out.println("top o' the stack");

   }

}
```

Runnable Example

```
public class MyRunnable implements Runnable {

    public void run()
    {
        go();
    }
    public void go()
    {
        doMore();
    }

    public void doMore()
    {
        System.out.println("top o' the stack");
    }
}
```

The class implements Runnable, so we have to define its run() method

Runnable Example

```
public class MyRunnable implements Runnable {

    public void run()
    {
        go();
    }
    public void go()
    {
        doMore();
    }

    public void doMore()
    {
        System.out.println("top o' the stack");
    }
}
```

All run() does is call another method go(), which in turn calls doMore(), which has a simple print statement.

Runnable Example

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```


Runnable Example

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```

Create an instance of this Runnable job.

Runnable Example

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```

Pass the job to a worker Thread object.

Runnable Example

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```

Start the thread – this will call the run() method of the Runnable object

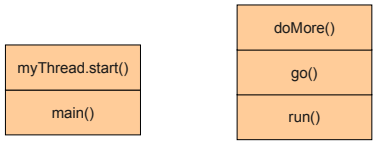
Runnable Example

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```

We'll also put a print statement here in the main method.

Multiple Threads



What gets printed if we run this?

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```


The Thread Scheduler

- We talked about threads being in different states, e.g. runnable and running
- The thread scheduler makes all the decisions about which thread moves from runnable to running, or when a thread leaves the running state
- **We do not control the scheduler**
- We do not control which thread runs when, nor how long it runs

The Thread Scheduler

- Because of this, we should never write code that depends on the scheduler working in a particular way
- We cannot assume that threadA will run to completion and then threadB will run to completion and so on
- There are some things we can do to affect which threads are run
  - e.g. putting a thread to sleep for a few milliseconds gives other threads a chance to run
  - More on that in a moment

So, what gets printed if we run this?

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```

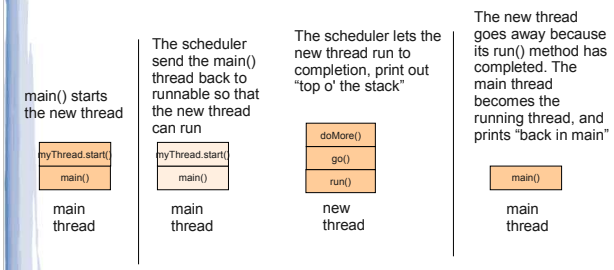
Answer: it depends

- It depends on the scheduler and how the scheduler decides what to run and when
- If you run this program multiple times, you are liable to get different results:

```
% java ThreadTester
   back in main
   top o' the stack
% java ThreadTester
   top o' the stack
   back in main
```

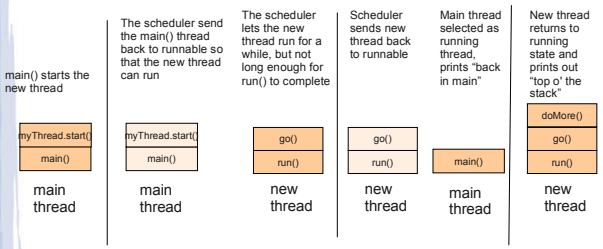

Why does it vary?

- Sometimes it runs like this:



Why does it vary?

- And sometimes it runs like this:



Thread Scheduler

- As mentioned, we can't assume that the new thread will be allowed to run to completion

Thread Priorities

- Programmers may assign priorities to threads
  - using `set/getPriority` methods and constants `MIN_PRIORITY, MAX_PRIORITY, NORM_PRIORITY` (range from 0 to 10)
- By default a thread gets the same priority as the thread that created it.
- In general, the running thread will be of equal or higher priority than the other threads in the *runnable* state but this isn't guaranteed!
- When all the threads in the *runnable* state have the same priority, the behaviour will depend on the way the scheduler is implemented.

Putting threads to sleep

- If we want to help our threads take turns, we can put them to sleep periodically
- We do this by calling the static sleep() method, indicating a duration in milliseconds
  - `Thread.sleep(2000)`
- This removes the thread from the running state
- It also means that the thread can't become the running thread again for at least 2 seconds

Putting threads to sleep

- Note: that doesn't mean the thread will become the running thread again in two seconds
- It means that after two seconds have elapsed, it will go back into the runnable() state and will wait to be chosen

Putting threads to sleep

- The sleep() method throws an InterruptedException
- It is rare that a thread will ever be interrupted from its sleep, but we still have to catch the exception

```
try {
   Thread.sleep(2000);}
catch (InterruptedException ex) {
    ex.printStackTrace();
}
```

```
public class MyRunnable implements Runnable {
public void run(){
  go();
}
public void go()
{
try {
  Thread.sleep(2000);}
catch(InterruptedException ex){
  ex.printStackTrace();}
doMore();
}
public void doMore()
{
System.out.println("top o' the stack");
}
}
```

In the go() method, let's add this sleep() call before doMore() gets called.

Now what gets printed if we run this?

```
public class ThreadTester {

public static void main(String[] args)
{
Runnable threadJob = new MyRunnable();
Thread myThread = new Thread(threadJob);
myThread.start();
System.out.println("back in main");
}
}
```


Putting threads to sleep

- We should now get a consistent ordering of our print statements

```
% java ThreadTester
        back in main
        top o' the stack
% java ThreadTester
        back in main
        top o' the stack
```


Putting threads to sleep

- We should now get a consistent ordering of our print statements

```
% java ThreadTester
        back in main
        top o' the stack
% java ThreadTester
        back in main
        top o' the stack
```

You will also see a pause of about 2 seconds before the second line is printed

Deadlock

- There are many pitfalls that one must be aware of when programming with threads.
- ***Deadlock*** is one of them.
- Deadlock occurs when two or more threads are unable to make progress because they are each waiting for each other to do something.

Busy Wait

- Example:

```
public class Worker extends Thread {
    private sometype result;

    public void run() {
     // Performs very long computation and sets result
    }

public static void main(String[] args) {
       Thread wt = new Worker();
       wt.start();
       while (wt.isAlive());
       System.out.println(result);
    }
}
```

Busy-Wait loop

Busy Wait (continued)

- Problems with the last example and a non-preemptive scheduler:
  - the main thread loops until the Worker thread completes;
  - the Worker thread cannot get the processor because the main thread is looping
- We must avoid writing code like this
  - result relies on the scheduler's decision; cannot show that it is correct
  - may not work for non-preemptive schedulers.
- So how do we fix it?

Thread Coordination using `sleep()`

- One way to fix the problem is to use the `sleep()` method to tell the scheduler to switch to another thread.
- `sleep(long n)` – stops running the current thread for (at least) n milliseconds
- `sleep()` is a static method of the `Thread` class
  - you can put to sleep only the current thread
- Sleep time is not accurate
  - when `sleep()` is executed the thread moves to the *waiting* state
  - when sleep time is over, it is moved back to the *runnable* state
  - thread can run any time after that.

Example using `sleep()`

```
public class Worker extends Thread {
    private sometype result;
    public void run() {
        // Performs very long computation and sets result     }

public static void main(String[] args) {
     Thread wt = new Worker();
     wt.start();
     while (wt.isAlive()) {
          try {
                Thread.sleep(800); // sleep for 800 msec
          }
          catch (InterruptedException e){
          ...
          }
     }
     System.out.println(result);
}
```


Example using sleep()

- Now the Worker thread has a chance to get the processor

Thread Coordination using `yield()`

- `yield()` is another static method of the `Thread` class
- When a thread executes `yield()` it is pre-empted and placed in the *runnable* state
  - preempted thread starts execution again when it is selected by the scheduler
  - it does not necessarily have to wait
- Example:
  - could replace `sleep(800)` with `yield()` in the previous program
    - we should remove the `try` block; `yield()` does not throw any exception
  - would this do what we want?

Example using `yield()`

```
public class Worker extends Thread {
    private sometype result;
    public void run() {
            // Performs very long computation and sets result
    }

    public static void main(String[] args) {
       Thread wt = new Worker();
       wt.start();

       while( wt.isAlive() )
          Thread.yield();

       System.out.println(result);
    }
}
```

Thread Coordination using `join()`

- `sleep()` and `yield()` usually require a loop that keeps checking a condition
  - waste processor cycles
- When the current thread `t` executes `r.join()` on another thread `r`, `t` will be placed in the *wait/blocked* state until `r` terminates
- The version `join(long n)` will make `t` wait **at most** `n` milliseconds
- `join()` throws an `InterruptedException` if the thread is interrupted
- Example: we can replace the last example's `while` loop with a call to `join()`.

Example using `join()`

```
public class Worker extends Thread {
    private  sometype  result;
    public void run() {
        // Performs very long computation and sets result     }

public static void main(String[] args) {
    Thread wt = new Worker();
    wt.start();
    try {
        wt.join();           // wait for wt to terminate
    }
    catch (InterruptedException e ) {
    ...
    }
    System.out.println(result);
}
```


In-Class Exercise II

- For the following two examples, indicate what you expect the output to be


```
public class MyRunner implements
Runnable

{

public void run()

{

System.out.println("This is
great");

Thread.yield();

go();

}


public void go()

{

System.out.println("having fun");

}
}
```

```
public class ThreadTester {


public static void main(String[]
args)

{

Runnable threadJob = new MyRunner();

Thread myThread = new
Thread(threadJob);

myThread.start();

System.out.println("back in main");

}

}
```

```
public class MyRunner implements
Runnable
{
public void run()
{
System.out.println("This is
great");
Thread.yield();
go();
}

public void go()
{
System.out.println("having fun");
}
}
```

```
public class ThreadTester {

public static void main(String[]
args)
{
Runnable threadJob = new MyRunner();
Thread myThread = new
Thread(threadJob);
myThread.start();
try{
myThread.join();
}
catch (InterruptedException ex){}
System.out.println("back in main");
}
}
```


Tea break!

Sharing Resources

- Many threads may need to access the same resource (object, file, memory, etc.). Such cases must be handled carefully.
- In the following (very contrived) example, we'll create a single bank account with an initial balance of $0 that will be shared by three threads.
- Each thread will deposit $100 to the account.
- We'll see that, unless we're careful, the account will not have a deposit of $300 by the time the three threads have finished running…

Sharing Resources

```
public class UnsyncAccount {
  private double balance;

  public UnsyncAccount() {
     balance = 0.0;
  }

  public void deposit(double amount) {
    double tempBalance = balance;
    // run some lengthy process here (or just sleep())
    balance = tempBalance + amount;
  }

  public double getBalance() {
    return balance;
  }
}
```

Example: A simple shared account ...

```
public class UnsyncDeposit extends Thread {
  private UnsyncAccount account;

  public UnsyncDeposit(UnsyncAccount a) {
      account = a;
  }

  public void run() {
      System.out.println("Thread " + this.getId()
          + " BEFORE deposit balance: "
          + account.getBalance());

      account.deposit(100);

      System.out.println("Thread " + this.getId()
          + " AFTER deposit balance: "
          + account.getBalance());
  }
```

All the run() method does is check the balance, try to deposit something, and check the balance again

Example: A simple shared account ...

```
public static void main(String[] args) {
  UnsyncAccount acc = new UnsyncAccount();
  Thread th1 = new UnsyncDeposit(acc);
  Thread th2 = new UnsyncDeposit(acc);
  Thread th3 = new UnsyncDeposit(acc);
  th1.start();   th2.start();   th3.start();
  try {
  th1.join();    th2.join();    th3.join();
  }
  catch (InterruptedException e){}
  System.out.println("Account balance is: " +
  acc.getBalance());
 }
}
```

We create three threads with a shared resource (a single account) and call start() on each of them. After they have completed, we print out the current balance of the account?

Example: A simple shared account ...

```
public static void main(String[] args) {
  UnsyncAccount acc = new UnsyncAccount();
  Thread th1 = new UnsyncDeposit(acc);
  Thread th2 = new UnsyncDeposit(acc);
  Thread th3 = new UnsyncDeposit(acc);
  th1.start();   th2.start();   th3.start();
  try {
  th1.join();    th2.join();    th3.join();
  }
  catch (InterruptedException e){}
  System.out.println("Account balance is: " +
  acc.getBalance());
 }
}
```

What value is printed out?
300 or **100**

Shared resources

- As you may have guessed by now, it depends

```
public void deposit(double amount) {
 double tempBalance = balance;
 // run some lengthy process here (or just sleep())
 balance = tempBalance + amount;
}
```

- (Again, this is a very contrived example, but illustrates something important)
- We set tempBalance to the current balance, say 0
- Then the scheduler selects some other thread to be running. In that thread, 100 more dollars is deposited.
- When this current thread gets back to the running state, it sets balance equal to tempBalance (still 0) and adds 100
- So we could get a balance of 100 when it should have been 200

Race Condition & Critical Sections

- In the previous example, the outcome depends on the way that the threads are scheduled to run. This is called a ***race condition***.
- To get correct results we need to ensure that the code that updates the account is executed by at most one thread at a time.
- Any code segment that must be run by only one thread at a time is called a ***critical section.***
- Any code segment that updates a resource that can be shared by multiple threads is a critical section.
- Java provides ***lock objects*** that can be used to tell the system that a section can be executed by only one thread at a time.

Lock Objects

- A lock object implements the `Lock` interface which is defined in the `java.util.concurrent.locks` package
- The **`Lock`** interface includes methods
  - `lock()` - if lock is available, it is acquired, otherwise wait
  - `unlock()` – releases the lock
- The same package has a number of classes implementing Lock.
- The most common is the **`ReentrantLock`** class which provides ***mutually exclusive*** or ***mutex locks***
  - only one thread can hold a given lock at a time

Using Locks

- Normally, a class whose objects are shared would declare a lock, say `myLock` and each critical section will be surrounded by calls to lock() and unlock():

```
myLock.lock();
critical section code
myLock.unlock();
```

- But if the critical section code throws an exception the lock will never be released. For that reason we always use the following:

```
myLock.lock();
try {
  critical section code
}
finally {
  myLock.unlock();
}
```

- So, the lock is always released (even if an exception is thrown)

Example: Bank Account with Lock object

```
public class SyncAccount {
  private double balance;

  private Lock lock = new ReentrantLock();

  public double getBalance() {
  return balance;
  }
```

Create a new Lock object

Example (cont'd)

```
public void deposit(double amount) {
lock.lock();
try {
double tempBalance = balance;
System.gc();  // run an expensive process
balance = tempBalance + amount;
}
finally {
lock.unlock();
}
}
```

We've now protected that critical section of code by locking it (and unlocking afterward) to ensure that only one thread at a time can run that section of code.

Example (cont'd)

```
public class SyncDeposit extends Thread{
  private SyncAccount account;

  public SyncDeposit(SyncAccount a) {
     account = a;
  }

  public void run() {
     System.out.println("Thread " + this.getId()
           + " BEFORE deposit balance: "
           + account.getBalance());
     account.deposit(100);
     System.out.println("Thread " + this.getId()
           + " AFTER deposit balance: "
           + account.getBalance());
  }
}
```

Again, we check the balance, add 100 dollars, and check the balance again.

Example (cont'd)

```
public static void main(String[] args) {
    SyncAccount acc = new SyncAccount();
    Thread th1 = new SyncDeposit(acc);
    Thread th2 = new SyncDeposit(acc);
    Thread th3 = new SyncDeposit(acc);

    th1.start();   th2.start();  th3.start();
    try {
          th1.join();  th2.join();  th3.join();
    }
    catch (InterruptedException e){}
    System.out.println("Account balance is: "
          + acc.getBalance());
}
}
```

What value is printed out?
100 or **300**

Example (cont'd)

- The appropriate use of a lock ensures that at most one thread can be running the critical section of code in the `deposit()` method at any given time.
- Now, every time we run this program, the balance on the account will be $300.

Synchronized Methods of Old Versions of Java

- Older versions of Java (prior to 1.5)do not have lock objects.
- Instead, every object has a lock that behaves like a ReentrantLock.
- If the lock is available, it is acquired when a synchronized method is called.
- A ***synchronized method*** is declared as

```
public synchronized void push(Object item)
 {
    // code for the method goes here
 }
```

and is synchronized on the lock of its implicit argument (**this**)

Synchronized Methods

- Synchronized instance methods allow at most one thread to run *any* of the object's synchronized methods at any time.
- Synchronized methods are simpler but less flexible.
- The `Account` class would be defined as follows if we use synchronized methods...

Account Example with Synchronized Methods

```
public class SyncAccount {
  private double balance;

  public synchronized void deposit(double amount) {
  double tempBalance = balance;
  System.gc(); // run an expensive process
  balance = tempBalance + amount;
  }

  public double getBalance() {
  return balance;
  }

}
```


Advanced Synchronization

- Suppose we have a producer that produces items for a consumer to consume.
- The producer and consumer will run on different threads
- The producer will place an item into a buffer where it will be retrieved by the consumer. The buffer will store only a single item
  - producer produces an item and places it in the buffer; the producer must wait if buffer is full (buffer stores only one item)
  - consumer removes the item from the buffer; the consumer must wait if the buffer is empty (there's nothing to consume)

Buffer Class: First attempt

```
public class BadBuffer {
   private int currentItem;
   private boolean full = false;
   private Lock lock = new ReentrantLock();

   public void add( int item ) {
      lock.lock();
      try {
         // wait if buffer is full
         while ( full ) ;
         currentItem = item;
         full = true;
      }
      finally {
         lock.unlock();
      }
   }
```


Buffer Class: First attempt (cont)

```
public int remove() {

   lock.lock();
   try {
      // wait if buffer is empty
      while ( !full ) ;
      full = false;
      return currentItem;
   }
   finally {
      lock.unlock();
   }
}
}
```


Deadlock Problems

- `BadBuffer` provides mutual exclusion :
  - add and remove cannot be executed at the same time
- But, there is a possibility for ***deadlock***
  - two or more threads are waiting for each other to release some locks; none can make any progress
- Suppose buffer is empty and consumer executes `remove()`. What will happen?
  -
  -
  -
- We need to do better

Synchronization Using Conditions

- To resolve this problem we should use ***condition objects***
- A condition object allows a thread to release a lock temporarily, so another thread can get that lock and run
- Each condition object belongs to a lock object and is created as follows:

```
Condition myCondition = lock.newCondition();
```

Synchronization Using Conditions

- A condition object implements the `Condition` interface that includes:
  - `await()`
    - the current thread releases the associated lock
    - the current thread moves to the *wait/blocked* state until another thread calls `signal()` or `signalAll()` on this condition
  - `signal()` or `signalAll()`
    - causes one or all of the threads that are blocked waiting on the condition to move to the *runnable* state
    - these threads will compete to get the lock again
    - one of them will get the lock and continue to run

Buffer Class: Using Conditions

```
public class GoodBuffer {
   private int currentItem;
   private boolean full = false;
   private Lock lock = new ReentrantLock();
   private Condition bufferEmpty = lock.newCondition();
   private Condition bufferFull = lock.newCondition();

   public void add( int item) {
      lock.lock();
      try {
         while ( full )         // wait for buffer to be empty
            bufferEmpty.await();
         currentItem = item;
         full = true;
         bufferFull.signalAll();   //notify consumers
      }
      catch (InterruptedException e) {
      }
      finally {
         lock.unlock();
      }
   }
```


Buffer Class (cont'd)

```
public int remove() {
   int returnValue = 0;

   lock.lock();
   try {
      // whait for buffer to be full
      while ( !full )
         bufferFull.await();
      full = false;
      returnValue = currentItem;
      bufferEmpty.signalAll();       //notify producers
   }
   catch (InterruptedException e) {
   }
   finally {
      lock.unlock();
   }

   return returnValue;
  }
}
```


Another deadlock example

- Imagine a simple BankAccount class with deposit() and withdraw() methods.

```
public void withdraw(double amount)

{

  balanceChangeLock.lock();

  try

   {

      while (balance < amount)

      . . .

   finally

{

      balanceChangeLock.unlock();
 }

}
```

Our BankAccount class has a lock because its methods access a shared resource, the balance. So in the withdraw method, we acquire the lock.

We put in a while loop to wait until the balance is sufficient to allow the withdrawal.

But how do we wait? If we put the thread to sleep, the lock will not be released and no threads will be able to call deposit because they will be unable to get the lock. We will be in a **deadlock** situation.

Condition Objects

- We can again use a condition object

```
public class BankAccount {

   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;
   private int balance;

   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      SufficientFundsCondition = balanceChangeLock.newCondition();
      . . .
   }
   . . .
}
```


Condition Objects

```
public void withdraw(double amount)
{
balanceChangeLock.lock();
try{
while (balance < amount) sufficientFundsCondition.await();

}
catch (InterruptedException ex){}

finally
{
balanceChangeLock.unlock();
}
}
```


Condition Objects

```
public void withdraw(double amount)
{
balanceChangeLock.lock();
try{
while (balance < amount) sufficientFundsCondition.await();

}
catch (InterruptedException ex){}

finally
{
balanceChangeLock.unlock();
}
}
```

When the balance is not sufficient, this thread temporarily releases its lock and goes into a **blocked** state. It waits for the balance to become sufficient.

It will know when the balance is sufficient because a signal will be sent to all threads currently being blocked as they await this condition.

In this case, that signal will be sent from the deposit method.

Condition Objects

```
public void deposit(double amount)
{
balanceChangeLock.lock();
try
{
 . . .
sufficientFundsCondition.signalAll();

}
finally {
balanceChangeLock.unlock();
}
}
```

A thread calling this method gets the lock, updates the balance, and notifies waiting threads that sufficient funds *may be* available now. Those threads become unblocked and can again compete to enter a running state.

Common Errors

- Calling await() without calling signalAll()
  - If a thread calls await() there needs to be a matching signalAll() that can be called by other threads, otherwise it will wait forever
- Calling signalAll() without locking the Object
  - A thread must own the lock that belongs to the condition object on which signalAll() is called. You'll get an exception otherwise.

Conclusion

- We've discussed how to build programs with multiple threads.
- To synchronize threads we can use Java's primitives:
  - lock objects
  - condition objects
- In CPSC 213 you will learn how to create independent processes on different processors and make them communicate/synchronize over a network

In-Class Exercise II

- given code with critical section, create lock, engage lock, run code, and make sure lock is always released

Learning Goals Review

- describe the multi-threaded programming model including thread scheduler, thread priority, and time slices.
- describe the various states that a Java thread can achieve and the events that lead to transition from one state to another
- define the terms deadlock, race condition and critical section
- identify possible legal traces of a multithreaded program
- identify deadlock and race conditions in a multithreaded program
- write a thread-safe code using Lock and Condition objects
- identify possible legal traces of a Java program that uses synchronization, locks and conditions

Exercises

Chapter 23, page 901
Exercises P23.1, P23.2, P23.7

Appendix: Main Methods of class Thread

- `public Thread()` --- Allocates a new Thread object
- `public Thread(Runnable target)`

- `public final boolean isAlive()` - Tests if this thread is alive
- `public static Thread currentThread()` - Get reference to currently executing thread

- `public final String getName()`
- `public final void setName(String name)`
- `public final int getPriority()`
- `public final void setPriority(int newPriority)`

- `public void start()` --- Causes thread to be scheduled; JVM calls its run() method
- `public void run()` --- If thread was constructed using a separate Runnable object, then that Runnable object's run method is called; otherwise, this method does nothing.
- `public void interrupt()` --- Interrupts this thread.
- `public final void join()` --- waits until the thread to which it is applied has died
- `public static void sleep(long millis)` --- puts currently executing thread to sleep
- `public static void yield()` --- currently executing thread is temporarily paused and allow other threads to execute