

# Programming 2 Revision

Jonathan Windle

University of East Anglia

*J.Windle@uea.ac.uk*

May 26, 2017

## 1 Threads

- Parallel structures
- Processes & Threads
- Concurrent Programming
- Java Threads
  - overview
  - Declaring Threaded Objects
  - Creating and Starting threaded objects
  - run() vs start()
  - Issues surrounding writing threaded code
- Thread Interaction 1
  - join
  - Example of join()
  - Thread interaction
  - Naively parallel algorithms
  - Parallel sort
- Sleep/Interrupt
- Thread Interaction 2

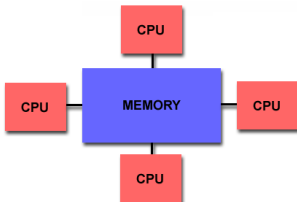
# Overview II

- Synchronization
  - Claiming monitor lock
  - Volatile Variables
  - Monitor Locks vs Semaphores
- Thread Interaction 3
  - Wait Notify
  - wait()
  - notify() and notifyAll()
  - Wait/Notify-Textual Example
  - Wait/Notify-Code example
  - Wait/Notify Common Problems
  - Wait vs Sleep
- Common Problems with Threads
  - Deadlock
  - Starvation
  - Livelock
- Summary

# Parallel Structures

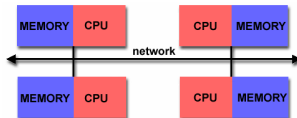
## Shared Memory

- Different processes access the same memory concurrently
- Each process shares memory with other processes.
- Structure adopted by multi core PC units now commonly available.



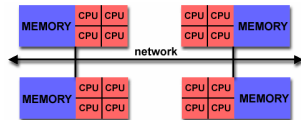
## Distributed Memory

- Each CPU has its own memory.
- Messages passed between processes to coordinate their output.
- Structure adopted by High-Performance Computing (HPC) facilities.



## Combined Shared/Distributed Memory

- Some CPUs share memory, some communicate via messages.
- Local processes also communicate with other groups via messaging.



# Processes & Threads

- When program is launched, a process is created.
- Processes can be sent to different processors by message passing scheme.
- A thread is a portion of a process that can run independently and concurrently with other portions of the process.
- Threads within a process can share data.
- Threads rely on the OS (or other program) to determine what processor they are sent to.

# Concurrent Programming

- Involves development of algorithms for running **simultaneously** in **multiple threads**.
- Java is inherently concurrent (Basic specification allows concurrent code development).
- C++ needs special library (e.g. pthreads) **C# is threaded**.
- Java does not allow for user controlled parallel processing.

# Overview

- Any Java program has at least one thread, **main**.
- Any Java thread of execution is associated with an instance of the **Thread** class.
- Before new thread can start, a new instance of the **Thread** class must be instantiated.
- A java thread class implements the **Runnable** interface. Therefore every **Thread** instance has a method:

```
|| public void run() {...}
```

- When the **Thread** is started the code in the body of the **run()** method is executed.

# Declaring Threaded Objects

## Extends Thread

```
public class MyThread extends Thread{
    public void run(){
        for(int i = 0; i < 13; i++){
            System.out.println
                ("Thread Iteration " + i);
        }
    }
}
```

- **Not preferred**, not defining the threads behaviour, just giving it something to run.
- **Cannot inherit** from this method.
- Each of your **thread creates unique object** and associate with it

## Implements Runnable

```
public class MyRunnable implements Runnable{
    public void run(){
        for(int i = 0; i < 13; i++){
            System.out.println
                ("Runnable Iteration " + i);
        }
    }
}
```

- **Preferred method** via **composition with Runnable**.
- **Extendible**, can have children classes.
- When you implement Runnable, it **shares the same object to multiple threads**.



# Creating and Starting threaded objects

## Starting Extends Thread

```
public static void main(){
    MyThread x = new MyThread();
    x.start();
}
```

- Call **start**, not **run**

## Starting Implements Runnable

```
public static void main(){
    MyRunnable x = new MyRunnable();
    (new Thread(x)).start();
}
```

- Create the runnable object and pass it to a new **Thread** and call **start** on that.

# run() vs start()

- Calling `run()` would just execute the code sequentially, essentially as a normal method.
- `start()` is a method defined in the `Thread` class and that calls `run()` method in separate threads that will run concurrently and thus thread the process.

# Issues surrounding writing threaded code

- Developing the algorithms can be tough.
- Controlling the interaction of different threads:
  - Pausing for other threads to finish (**join**).
  - Pausing for a set time or until an exception occurs (**sleep/interrupt**).
  - Waiting for another thread to notify that it's ok to continue (**wait/notify**).
- Controlling access to shared memory (**synchronize**).

# Join()

Used to wait for another thread to finish

```
|| void join()
```

- This thread would wait for the calling thread to die before continuing.

```
|| void join(long millis)
```

- Waits at most, millis milliseconds for the thread to die.

```
|| void join(long millis, int nanos)
```

- Waits at most millis milliseconds plus nanos nanoseconds for the thread to die.

If two threads are waiting for the other to finish the process then it will never finish.

# Example of join()

Code where the `void main()` method waits for two threads.

```
public static void main(string[] args) {
    MyThread t1 = new MyThread("Thread 1",1);
    MyThread t2 = new MyThread("Thread 2",2);
    t1.start();
    t2.start();
    try{
        t1.join();//Main now waits for t1 to die
        t2.join();//Main now waits for t2 to die
    }catch(InterruptedException ex){
        System.exit(0);
    }
    /*****
    * This for loop will only start when t1 and t2 have finished executing
    *****/
    for(int i = 0; i < 100000; i++) {
        if(i%1000==0)
            System.out.println("Main thread iteration: " + i);
    }
}
```

The `main` thread has been forced to wait for `t1` and `t2` to finish, but `t1` and `t2` are still running concurrently and therefore run completely independently.

Enforces sequential behaviour on concurrent code

# Thread interaction

Threads can **interact** with each other either through:

- **Storing references to each other** (and calling methods on those references)
- **Accessing shared memory**: synchronisation.
  - **Through the use of static variables within a threaded class.**

```
public class MyThread {  
    public static ArrayList<Integer> sharedMem = new ArrayList<>();  
  
    public void run() {  
        for(int i = 0; i < 400; i++) {  
            if(!sharedMem.contains(i))  
                sharedMem.add(i);  
        }  
    }  
}
```

The static variable, **sharedMem** is **accessible** through any instance of **MyThread**, if multiple instances of **MyThread** exist and **start()** is called in quick succession, then **all** in this case will interact with **sharedMem** in a **similar way**.

# Naively parallel algorithms

- If an algorithm can be split into sub-problems, then recombine the results from sub-problems, a thread system could be applied.
- If the threads do not interact with each other, then it is said that the problem is **Naively parallel**.

Concurrent sort(comparable[] ar, int n)

- 1 Split array into subarrays.
- 2 Create a thread to sort each subarray.
- 3 Merge sub arrays into sorted array.

# Parallel sort

```
public class ThreadSort extends Thread {  
    public static Comparable data[]; //Shared across all threads  
    int start; //Local start position  
    int end; //Local end position  
  
    public ThreadSort(int s, int e) {  
        start = s;  
        end = e;  
    }  
  
    public void run() {  
        Arrays.sort(data, start, end);  
    }  
}  
  
public static void main(String [] args) {  
    int m = 100;  
    int n = 10;  
    Comparable[] d = new Comparable[m];  
    ThreadSort[] arr = new ThreadSort[n];  
    for(int i = 0; i < n; i++) {  
        arr[i] = new ThreadSort(i*m/n, (i+1)*m/n);  
        arr[i].start();  
    }  
    //Main should join to ensure all sub sorts are complete before executing merge loop  
    for(int i = 0; i < n-1; i++)  
        merge(d, 0, (i+1)*m/n, (i+1)*m/n, (i+2)*m/n);  
}
```



# Sleep/Interrupt

- `sleep()`
  - This pauses the thread for a fixed time or an `InterruptedException` is thrown when another thread has interrupted the current thread.
- `interrupt()`
  - Interrupts the thread called on. Uses an internal flag called the `interrupt status`. Calling the `interrupt` method sets the flag.
  - This only sets the internal flag, it doesn't stop the thread, or do anything if the thread is not sleeping.

# Synchronization

- **Synchronization** involves **locking an object** so that it can **only be accessed by one thread at a time**.
- Allows **enforcing mutual exclusion** between threads.
- **Mutual exclusion** is **implemented through monitor locks**. All objects in java **have a monitor lock** that can be owned by other objects.
  - If **one object owns the monitor lock** of another object, then **it has exclusive access to that object**, until it gives up the lock.
- **Synchronization** is **enforced using the synchronized keyword**. Any **synchronized method or statement** can **only be accessed by one thread at any time**.
- **When a thread invokes a synchronized method**, it **automatically acquired the monitor lock** for that method's object and release it when the method returns.

# Claiming monitor lock

- **Direct synchronized block:**

```
private SharedMemClass names;  
public void run() {  
    synchronized(names){ //Nothing else can touch names until after this block  
        names.alter();  
    }  
}
```

- **synchronized methods:**

```
public class SharedMemClass {  
    synchronized public void alter(){} //Synchronized method  
}  
public class Modifier extends Thread() {  
    private SharedMemClass names;  
    public void run() {  
        //Instance of Modifier claims Monitor lock because of synchronized method  
        names.alter();  
    }  
}
```

## block vs methods

Sometimes a method only needs mutual exclusion for a part, such as to read the memory, then update the memory. Only updating the memory needs to be synchronized via a block, opposed to the whole method.

# Claiming monitor lock cont.

- **Direct class monitor lock:**

```
public class SharedMemClass {
    synchronized public void alter(){} //Synchronized method
    public static ArrayList<String> str = new ArrayList<>();
}
public class Modifier extends Thread() {
    private SharedMemClass names;
    public void run() {
        //This instance of Modifier claims lock on the whole class, locking all static variables
        //of the class
        synchronized(SharedMemClass.class);
    }
}
```

- **Synchronized static methods:**

```
public class SharedMemClass {
    synchronized static public void alter(){} //Synchronized method
    public static ArrayList<String> str = new ArrayList<>();
}
public class Modifier extends Thread() {
    private SharedMemClass names;
    public void run() {
        //This instance of Modifier claims lock on the whole class, locking all static variables
        //of the class, because alter() is static and synchronized
        SharedMemClass.alter();
    }
}
```

# Volatile Variables

- **Atomic** actions are actions that occur all at once and have no chance of being interrupted or partially complete.
- This means it is impossible for **threads** to interfere with each other when performing **atomic** actions.
- In Java read and write operations on a single variable can be made **atomic** by using the access modifier **volatile**.
- **volatile** variables act as though it is enclosed in a **synchronized** block, **synchronized** on itself.
- **volatile** variables are often used to signal between **threads**.
- All operations on the variable, happen straight to the variable, it is never cached locally to a thread.

Characteristic	Synchronized	Volatile
Type of variable	Object	Object or primitive
Null allowed?	No	Yes
Can block?	Yes	No
All cached variables synchronized on access?	Yes	From Java 5 onwards
When synchronization happens	When you explicitly enter/exit a synchronized block	Whenever a volatile variable is accessed.
Can be used to combined several operations into an atomic operation?	Yes	Pre-Java 5, no. Atomic get-set of volatiles possible in Java 5.

# Monitor Locks vs Semaphores

- When a **thread** enters a **synchronized** statement, it owns the **monitor lock** for the object.
- **monitor locks** restrict access to an object to one single thread.  
**semaphores** can restrict access to an object to more than one thread.

# Wait Notify

- Sometimes want a **thread** to **wait** for something in another object to happen before continuing.
- A **thread** can wait for an object to **notify** when something occurs.
- This can **only** be the case if the **thread** owns a **monitor lock** on the object it is waiting to be notified by and therefore **must occur in a synchronized block**.

```
if(something) //Check condition before waiting.  
    someObject.wait();  
//Continues from here after notify  
doSomethingElse();
```

- This means that the **calling object** will **wait for someObject to call notify** before continuing.

- The calling thread waits for an event to occur in the object called upon. The event could be from another thread or some other operating system event.
- Throws an InterruptedException if interrupted and not notified.
- Can be timed passing a long, similar to sleep.
- Must be in a synchronized block.



# notify() and notifyAll()

- `notify()`
  - Notifies one random thread. No control in which one.
- `notifyAll()`
  - Notifies all waiting threads. No reason can be given for the notification.
- Both must be in synchronized blocks.

# Wait/Notify-Textual Example

## Producer

Creates goods at random intervals and notifies consumers.

- 1 Sleep for random interval.
- 2 Make a product.
- 3 Notify any thread waiting.

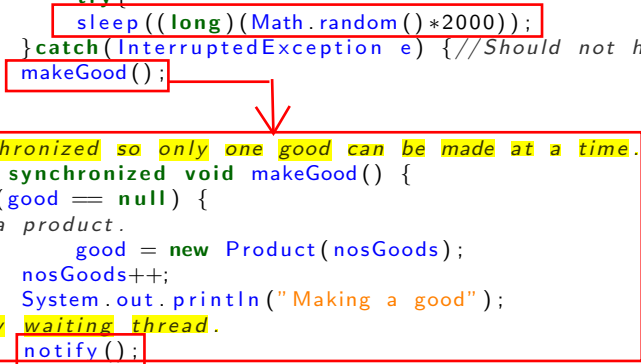
## Consumer

Waits for a producer to make a good, then purchase it.

- 1 Synchronize on Producer.
- 2 Wait for a notification.
- 3 Buy Product.

# Wait/Notify-Code example - Producer

```
public class Producer extends Thread {
    Product good=null;
    int nosGoods = 0;
    public void run() {
        while(!endOfSimulation()) {
//1) Sleep for random interval.
            try{
                sleep((long)(Math.random()*2000));
            }catch(InterruptedException e) {//Should not happen}
            makeGood();
        }
    }
//Synchronized so only one good can be made at a time.
    public synchronized void makeGood() {
        if(good == null) {
//2) Make a product.
            good = new Product(nosGoods);
            nosGoods++;
            System.out.println("Making a good");
//3) Notify waiting thread.
            notify();
        }
    }
}
```

A red box highlights the `makeGood()` method call in the `run()` loop. A red arrow points from this box to another red box that encloses the entire `makeGood()` method definition. This visualizes that the `makeGood()` method is synchronized, ensuring mutual exclusion.

# Wait/Notify-Code example - Consumer

```
public class Consumer extends Thread {
    Producer p;
    ArrayList<Product> goods;
    public Consumer(Producer p) {
        this.p = p;
        goods = new ArrayList<>();
    }
    public void run() {
        while(!p.endSimulation()) {
            if(p.hasGood()) {
                //1) Synchronize Producer
                synchronize(p){
                    try{
                        //2) Wait for notification
                        p.wait();
                    } catch(InterruptedException e){//Shouldn't
                        happen}
                }
            }
            //3) Buy Product.
            goods.add(p.buy());
        }
    }
}
```

# Wait/Notify Common Problems

- You need to **check the condition** before waiting otherwise **risk of never being notified**.
- **Waking** from wait **doesn't mean the condition** waiting for **has been satisfied**, **may need a loop of waiting** to condition has definitely been satisfied.
- The timed **wait**, **waits forever if given 0 as a time**.
- Calling **wait releases lock on object being waited on**, but **not on other objects** locked by the class:

```
synchronized (object1);  
synchronized (object2) {  
    object2.wait(); //object2 lock released. object1 still  
                    locked  
}
```

# Wait vs Sleep

- `Thread.sleep()` is a `static` method that `sends thread` into the "not runnable" state for a set time.
- `lock.wait()` is called on an `Object` (lock), **not** a `thread`.
  - This causes the `current thread` to `wait for the object`. **It does not cause lock to wait.**
- Before `lock.wait()` is called, the `thread must synchronize` on the `lock object`.
- `lock.wait()` releases the `lock on the object` and `adds the thread to the "wait list" associated with lock`.
- Another `thread` can `synchronize the same lock object` and call `lock.notify()` that `wakes up the original, waiting thread`.
- Essentially `wait()/notify()` is like `sleep()/interrupt()`, only the `active thread` does not need a direct reference to the sleeping thread, but `only to the shared lock object`.

# Deadlock

- **Deadlock** describes a situation where two threads are waiting for each other to finish and never complete.
- **synchronization** is vital but it can lead to deadlock.
  - 1 **A starts** executing and **is locked**.
  - 2 **B starts** executing and **is locked**.
  - 3 **A calls B** and **waits for B to finish**
  - 4 **B calls A** and **waits for A to finish**

# Starvation

- **Starvation** is where a **thread** is continually denied access to resources and is therefore **unable to progress**.
- This happens when **"greedy" threads take resources for long periods**.
- Essentially a **DDOS** on a **thread** and can **drastically slow down a program**.
- Can be **caused by threads being given a high priority**. A deterministic priority queue can cause **starvation**.



- **Threads** often act in response to an action of another thread.
- If the other **threads** action is also a **response to the action of another thread**, then **livelock** may occur.
- Unable to make progress.
- In this case the **threads** are not blocked, just responding to each other in a loop.
- Comparable to the awkward two people passing in a corridor going the same way each time.

# Summary

- Made threaded by `extends Thread` or `implements Runnable`.
- `run()` methods run concurrently.
- Threads `claim monitor lock` through keyword `synchronized`.
- Threads can effect each others run time behaviour through the `wait/notify` && `sleep/interrupt`. `wait/notify` requires `synchronization`.
- Presents new potential problems

# The End