Threads/Concurrency

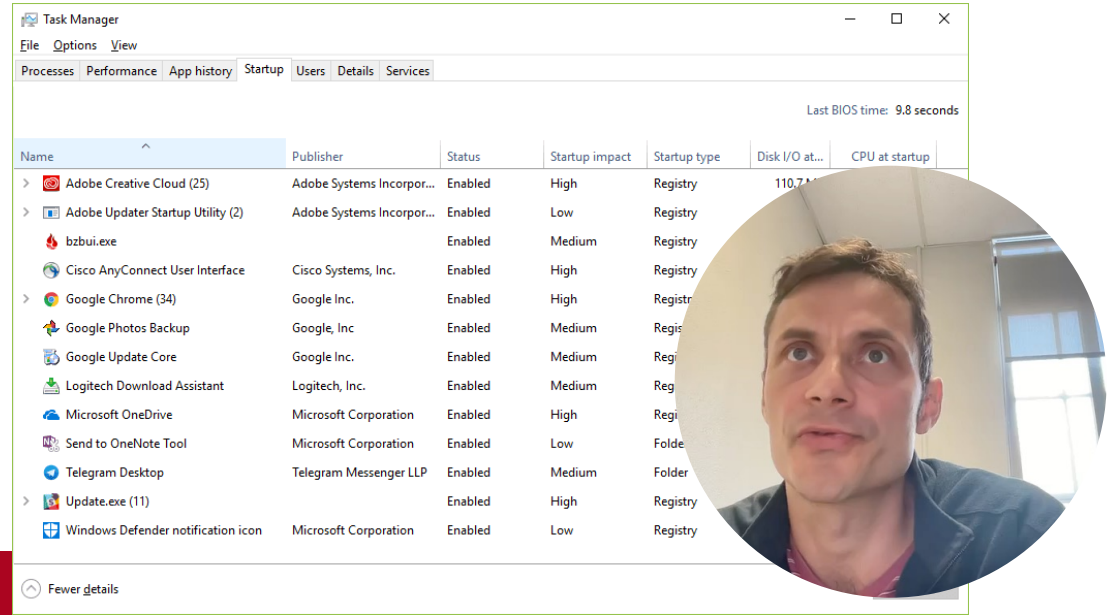
Concurrency

- In concurrent programming, there are two basic units of execution: ***processes*** and ***threads***.
- In the Java programming language, concurrent programming is mostly concerned with **threads.**
  - *However, processes are also important.*

Concurrency in general

- Operating system does not run one application at the time, but multiple in parallel.

- A computer system normally has many active processes and threads.

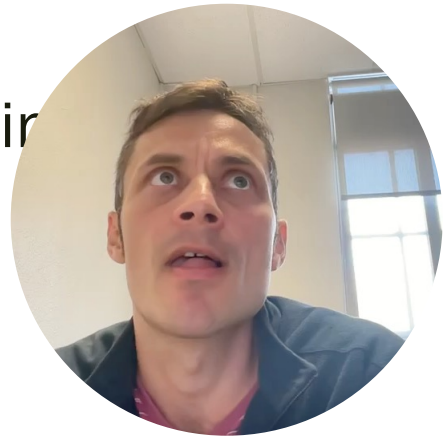


Processes

- A process has a self-contained execution environment.
  - ***a complete, private set of basic run-time resources;*** *in particular, each process has its* ***own memory space.***

- Processes synonymous with programs or applications.

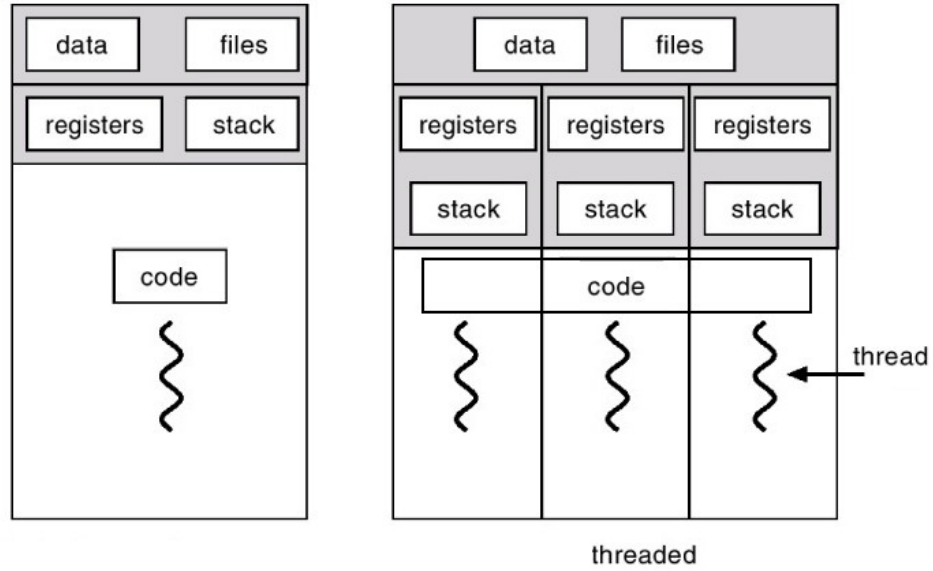
- A single application may be a set of cooperating processes.
- Most implementations of the Java virtual machine run as a sin
  process.



Communication between processes

- *Inter Process Communication* (IPC) resources,
  - *such as pipes (operating systems)*
  - *or sockets (later lecture).*

Threads



- *Lightweight processes*.

- Both processes & threads provide an execution environment.
  - *Creating a new thread requires fewer resources! (faster)*

- Threads exist within a process
  - at least one thread per process.
  - threads **share the process's resources,**
  - e.g. **memory & open files.**

Threads

- Efficient, but potentially problematic, communication.
- Multithreaded execution is an essential feature of the Java platform.

— whether you like it or not, this is the future!

- Every app has 1 or more threads.

— all starts with one thread, called the ***main thread***.

— it has the ability to create additional threads.

Threads in Java

- Two approaches both have a method run()

- `Thread` class extension
- `Runnable` interface implementation

Example

Both examples invoke Thread.start in order to start the new thread.

```
public class HelloThread extends Thread {
  public void run() {
        System.out.println("Hello from a thread!");
  }
 }
```

```
public class HelloRunnable implements Runnable {
  public void run() {
        System.out.println("Hello from a thread!");
  }
}
```

Example

Both examples invoke Thread.start in order to start the new thread.

```
public class HelloThread extends Thread {
  public void run() {
        System.out.println("Hello from a thread!");
    }
  public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

```
public class HelloRunnable implements Runnable {
  public void run() {
        System.out.println("Hello from a thread!");
    }

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

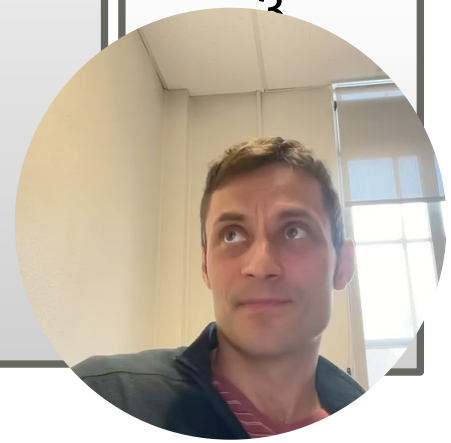
THE UNIVERSITY OF ARIZONA

Two threads

How many threads?

```
class TestThreads extends Thread {
 public void run() {
   for (int i=1;i<=5;i++) {
     try{
        Thread.sleep(500);
     } catch(InterruptedException e) {
        System.out.println(e);
     }
   System.out.println(i);
 }
}
 public static void main(String args[]){
  TestThreads t1= new TestThreads();
  TestThreads t2= new TestThreads();

 t1.start();
 t2.start();
 }
}
```

```
1
1
2
2
3
```



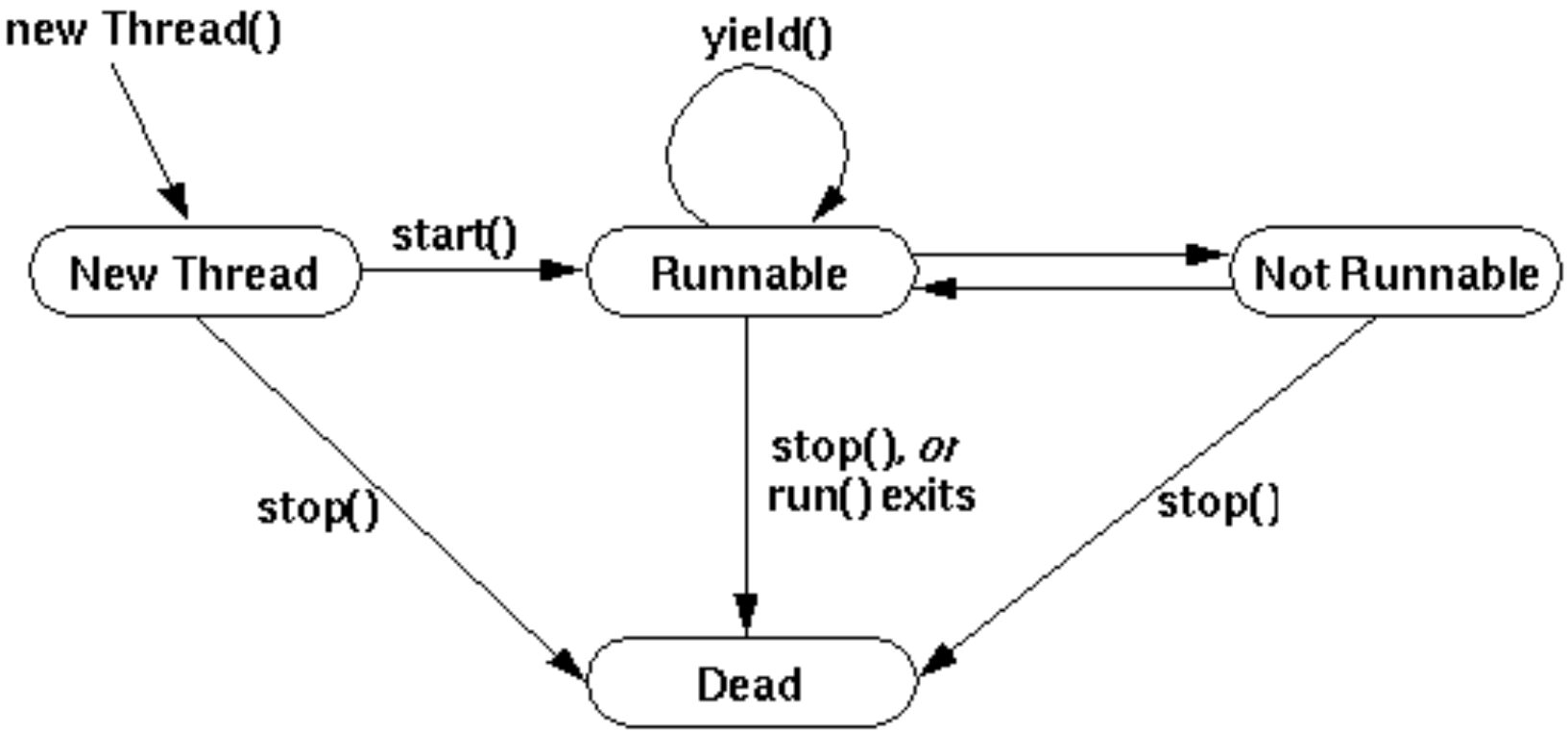
Two threads

How many threads?

```
class TestThreads extends Thread {
 public void run() {
   for (int i=1;i<=5;i++) {
     try{
        Thread.sleep(00000000);
     } catch(InterruptedException e) {
        System.out.println(e);
     }
   System.out.println(i);
 }
}
 public static void main(String args[]){
  TestThreads t1= new TestThreads();
  TestThreads t2= new TestThreads();

 t1.start();
 t2.start();
 }
}
```

```
1
2
3
1
2
```

Stop




```
Thread myThread = new MyThreadClass();
myThread.start();
try {
    Thread.currentThread().sleep(10000);
} catch (InterruptedException e){
}
myThread.stop();
```

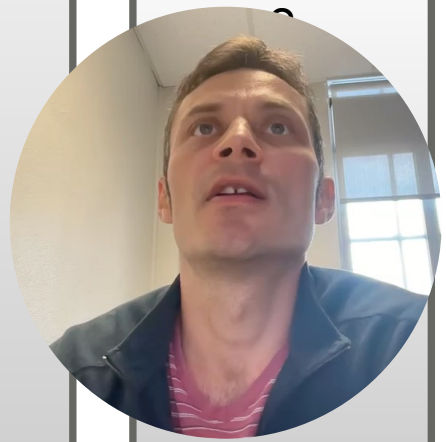
Join

The join() method waits for a thread to die.

In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

```
class TestThreads extends Thread {
 public void run() {
   for (int i=1;i<=5;i++) {
     try{
        Thread.sleep(500);
     } catch(InterruptedException e) {
        System.out.println(e);
     }
     System.out.println(i);
   }
}
 public static void main(String args[]){
  TestThreads  t1=new  TestThreads();
  TestThreads  t2=new  TestThreads();
  TestThreads  t3=new  TestThreads();
   t1.start();
  try{
    t1.join();
  }catch(Exception e){System.out.println(e);}
 t2.start();
 t3.start();
}
}
```

1
2
3
4
5
1
1
2



Thread Interference

Class Counter
designed so that increment adds 1 to c, and decrement subtracts 1 from c.

However, if a Counter object is referenced from multiple threads, interference between threads may lead to unexpected results!

...

```
class Counter {
    private int c = 0;

    public void increment() {
            c++;
      }

    public void decrement() {
            c--;
      }

    public int value() {
            return c;
      }

}
```

Thread Interference

```
class Counter {
    private int c = 0;

    public void increment() {
            c++;
       }

    public void decrement() {
            c--;
       }

    public int value() {
            return c;
       }

}
```

…

Interference happens when two operations, running in different threads, act on the same data, *interleave*.

Two operations consist of multiple steps, and the sequences of steps overlap.

increment

```
public class GreedyThread extends Thread {
  private Counter counter;
  public GreedyThread(Counter counter) {
       this.counter = counter;
  }
  public void run() {
      for(int i=0;i<1000;++i)
            counter.increment();
  }
}
```

decrement

```
public class FixerThread extends Thread {
  private Counter counter;
  public FixerThread(Counter counter) {
       this.counter = counter;
  }
  public void run() {
      for(int i=1000;i>0;--i)
            counter.decrement();
  }
}
```

Thread Interference

```
public class Testee {
   public static void main(String[] args) {

        Counter counter = new Counter();
        Thread t1 = new FixerThread(counter);

        Thread t2 = new GreedyThread(counter);

        t1.start();t2.start();
        try {
                t1.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        try {
                t2.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        System.out.println(">>"+ counter.value());
  }
}
```

Thread Interference

```
public class Testee {
   public static void main(String[] args) {

        Counter counter = new Counter();
        Thread t1 = new FixerThread(counter);

        Thread t2 = new GreedyThread(counter);

        t1.start();t2.start();
        try {
                t1.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        try {
                t2.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        System.out.println(">>"+ counter.value());
  }
}
```

```
$>>-114
```

Thread Interference

```
public class Testee {
   public static void main(String[] args) {

        Counter counter = new Counter();
        Thread t1 = new FixerThread(counter);

        Thread t2 = new GreedyThread(counter);

        t1.start();t2.start();
        try {
                t1.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        try {
                t2.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        System.out.println(">>"+ counter.value());
  }
}
```

```
$>>-114
```

```
$>>17
```

Thread Interference

```
public class Testee {
   public static void main(String[] args) {

        Counter counter = new Counter();
        Thread t1 = new FixerThread(counter);

        Thread t2 = new GreedyThread(counter);

        t1.start();t2.start();
        try {
                t1.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        try {
                t2.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        System.out.println(">>"+ counter.value());
  }
}
```

```
$>>-114
```

```
$>>17
```

```
$>>201
```

Thread Interference

```
public class Testee {
   public static void main(String[] args) {

        Counter counter = new Counter();
        Thread t1 = new FixerThread(counter);

        Thread t2 = new GreedyThread(counter);

        t1.start();t2.start();
        try {
                t1.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        try {
                t2.join();
        } catch (Exception e) {
                System.out.println(e);
        }
        System.out.println(">>"+ counter.value());
   }
}
```

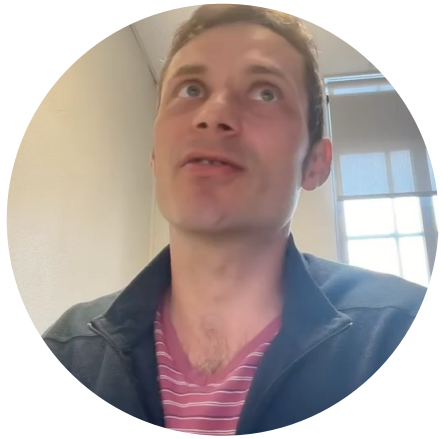
$>>-114

$>>17

$>>201

Memory consistency errors.

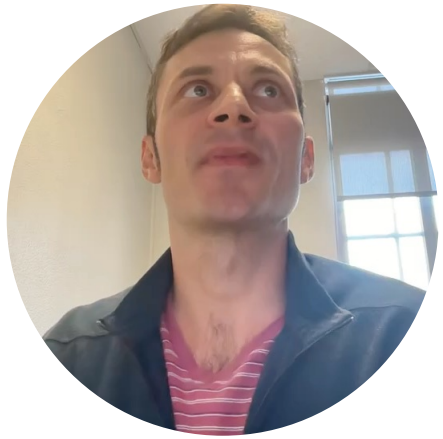
- *occur when different threads have inconsistent views*
- *of what should be the same data.*



Memory consistency errors.

- The key to avoiding memory consistency errors is understanding the ***happens-before*** **relationship**

  – *This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement.*



Memory consistency errors.

- Suppose a simple int field is defined and initialized:
  - `int counter = 0;`
- The counter field is shared between two threads, A and B.
  - *Suppose thread A increments counter:*
  - `counter++;`
- Then, shortly afterwards, thread B prints out counter
  - `System.out.println(counter);`

Memory consistency errors.

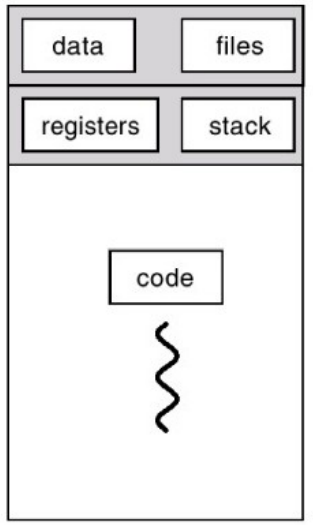
If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1"

But if they are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a **happens-before** relationship between these two statements.

- Locks!

Solving consistency errors

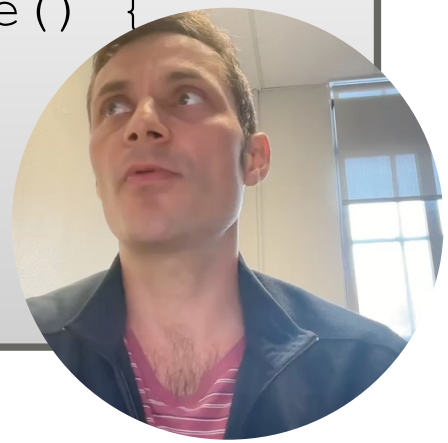
1. Thread *synchronized methods*
2. Thread *synchronized statement*
3. *Atomic access*



Thread *synchronized methods*

```
class SyncCounter {
    private int c = 0;

    public synchronized void increment() {
          c++;
      }

    public synchronized void decrement() {
          c--;
      }

    public synchronized int value() {
          return c;
      }

}
```

$>>0



Thread *synchronized methods*

synchronized methods has two effects:

First, it is not possible for two invocations of synchronized methods on the same object to interleave.

- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object blocks (suspends execution) until the first thread is done with the object.

Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object.

- This guarantees that changes to the state of the object are to all threads.

Thread *synchronized methods*

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock **for that method's object** and releases it when the method returns.

- Java does the dirty work for you!
- Not the best performance!

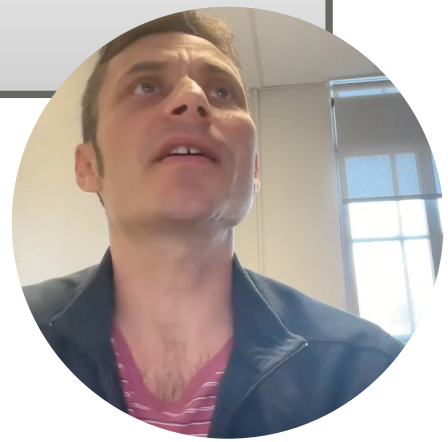
The lock release occurs even if the return was caused by an uncaught exception.

Thread *synchronized statement*

Synchronized statements **must specify the object that provides the intrinsic lock**

The addName method needs to synchronize changes to lastName and nameCount, but also needs to avoid synchronizing invocations of other objects' methods.

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```



Thread *synchronized statement*

Fine-grained synchronization (better performance).

Suppose, that class SynCounter has two instance fields, c1 and c2, that are never used together.

All updates of these fields must be synchronized,

- but there's no reason to prevent an update of c1 from being interleaved with an update of c2
- and doing so reduces concurrency by creating unnecessary blo

Instead of using synchronized methods or otherwise using the associated with this, we create two objects solely to provide loc

Thread *synchronized statement*

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

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Thread *synchronized statement*

Suppose, that
class SynCounter has
two instance fields,
c1 & c2, that are never
used together.

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

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Atomic Access

In programming, an *atomic* action is one that effectively happens all at once

An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all

No side effects of an atomic action are visible until the actio

Atomic Access

- We have already seen that an increment expression, such as count++, does not describe an atomic action.
  - *Even very simple expressions can define complex actions that can decompose into other actions.*
  - *However, there are actions you can specify that are atomic.*

Atomic Access

- *Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).*
- *Reads and writes are atomic for all variables declared* `volatile` *(including long and double variables)*

- *double/long reads in two steps! 64bit ~ write/read 32 bits at the time*
- *Thread could possibly see 32bits only*
- *Informs compiler about concurrent environment to not to do optimization*

https://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html

Example

```
private boolean isActive = thread;
public void printMessage() {
  while (isActive) {
     System.out.println("Thread is Active");
   }
}
```

Without *volatile* variable it may result in an infinite loop. Without the *volatile modifier*, it is not guaranteed that one Thread sees the updated value of isActive from other Thread.

The compiler is **may cache value of isActive instead of reading it from main m every iteration.**

By making isActive a volatile variable you avoid these issue.

Atomic Access

- Atomic actions cannot be interleaved, so they can be used without fear of thread interference.
- Using **volatile** variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.
  - *This means that **changes to a volatile variable are visible to other threads.***

ISSUES WITH CONCURRENCY

Deadlock

Starvation

Livelock

Poor synchronization impacts performance a lo

Deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.

Starvation

- *Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
- This happens when shared resources are made unavailable for long periods by "greedy" threads.
  - *E.g. I need 3 resources at once.. I have 2..*

Livelock

- A thread often acts in response to the action of another thread.
- If the other thread's action is also a response to the action of another thread, then *livelock* may result.
  - *As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work.*

Livelock

- Example:

– *Two people are attempting to pass each other in a corridor:*
- Alphonse moves to his left to let Gaston pass,
- while Gaston moves to his right to let Alphonse pass.
- Seeing that they are still blocking each other,
- Alphone moves to his right, while Gaston moves to his left.
- They're still blocking each other, and so on…

COORDINATE THREADS TO WORK TOGETHER

How to coordinate threads to work together

- a *polling?*
- *Why 1000?*

```
public synchronized void guardedJoy() {
  while(!joy) {
      try {
          Thread.sleep(1000);
      } catch (InterruptedExcept
  }
  System.out.println("Joy achie
}
```

How to coordinate threads to work together

- a *guarded block*

```
public synchronized void guardedJoy() {
  while(!joy) {
     try {
          wait();
    } catch (InterruptedExcept
  }
  System.out.println("Joy achie
}
```

Guarded Blocks

- The most common coordination idiom.
- Such a block begins by polling a condition that must be true before the block can proceed.
  - *There are a number of steps to follow in order to do this correctly.*
- The invocation of wait() does not return until another thread has issued a notification() that some special event may have occurred — though not necessarily the event this thread is waiting for

```
public synchronized void guardedJoy() {
  while(!joy) {
     try {
          wait();
    } catch (InterruptedExcept
  }
  System.out.println("Joy achie
}
```

Guarded Blocks

- The most common coordination idiom.
- Such a block begins by polling a condition that must be true before the block can proceed.
  - *There are a number of steps to follow in order to do this correctly.*
- The invocation of wait() does not return until another thread has issued a notification() that some special event may have occurred — though not necessarily the event this thread is waiting for

```
public synchronized notifyJoy() {
    joy = true;
    notifyAll();//all threads
    // notify()
}
```

```
public synchronized void guardedJoy() {
  while(!joy) {
     try {
        wait();
     } catch (InterruptedExcept
  }
  System.out.println("Joy achie
}
```

The most common example ~ Producer/Consumer

- There are many interesting situations where separate concurrently running threads do share data and must consider the state and activities of those other threads.
- One such set of such situations are known as Producer/Consumer scenarios
- the Producer generates a stream of data
- which then is consumed by a Consumer.

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

Sample situations

- Producer

- Limited storage

- Consumer

2 examples

1 slot

Showed in slides!

10 slots available and faster producer

Left as assignment ;)

Testee

```
class ProducerConsumerTest {
  public static void main(String args[]) throws InterruptedException {
            CubbyHole c = new CubbyHole();
            Producer p1 = new Producer(c, 1);
            Consumer c1 = new Consumer(c, 1);
            p1.start();
            c1.start();
  }
}
```

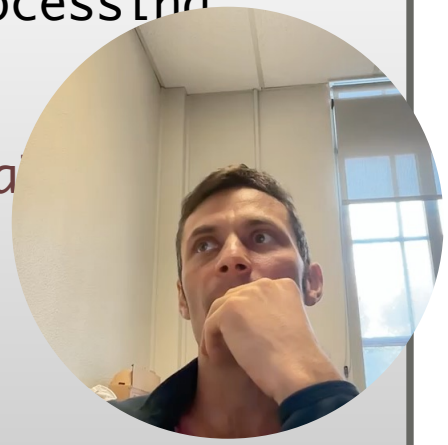
```
class CubbyHole {
        private int seq;
        private boolean available = false;
        public synchronized int get() {
                while (available == false) {
                        try {
                                wait();
                        } catch (InterruptedException e) {..}
                }
                available = false;
                notify();
                return seq;
        }
        public synchronized void put(int value) {
                while (available == true) {
                        try {
                                wait();
                        } catch (InterruptedException e) {..}
                }
                seq = value;
                available = true;
                notify();
        }
}
```

Example app Producer

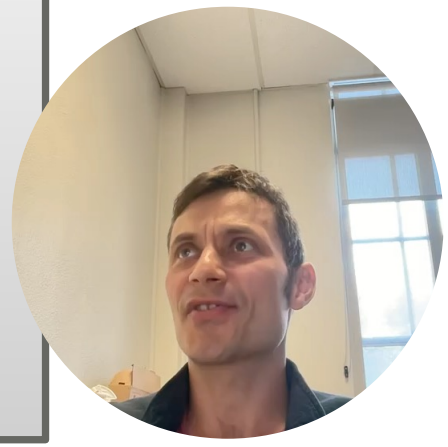
```
class Producer extends Thread {
      private CubbyHole cubbyhole;
      private int number;

      public Producer(CubbyHole c, int number) {
            cubbyhole = c;
            this.number = number;
      }
      public void run() {
        for (int i = 0; i < 100; i++) {
          cubbyhole.put(i);
          System.out.println("Producer #"+this.number+" p
        }
      }
}
```

Example app Consumer

```
class Consumer extends Thread {
        private CubbyHole cubbyhole;
        private int number;
        public Consumer(CubbyHole c, int number) {
                cubbyhole = c;
                this.number = number;
        }
        public void run() {
          int value = 0;
          for (int i = 0; i < 100; i++) {
            try {
                sleep((int) (Math.random() * 100));// simulate processing
            } catch (InterruptedException e) {..}
            value = cubbyhole.get();
            System.out.println("Consumer#"+this.number+" got:”+va
          }
        }
}
```



```
class CubbyHole {
        ??

        Queue

        • Remember to use all 10 slots?
        • Starts at 10 empty slots
        • get() increments add put() decrements empty slots
        • Do not let the empty slots to go under 0 - wait
}
```



Really cools staff in Java

- Executor
  - *Creating new thread is time consuming, can we reuse threads?*

Executor

- It is better to separate thread management & creation from the rest of the app.
- Objects that encapsulate these functions are known as *executors*.
- Java Executor is a simple interface that supports launching new tasks.

Executor

If we have a Runnable object `runnable`, and an executor object we can replace

```
(new Thread(runnable)).start();
```

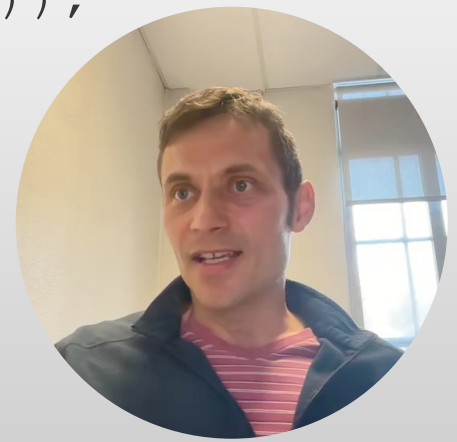
with

```
executor.execute(runnable);
```

- newSingleThreadExecutor creates an executor that executes a single task at a time
- **newFixedThreadPool** create an executor that uses a fixed thread pool
- newCachedThreadPool creates new threads as needed, & reuse previ
  threads if available

Executor

```
class NetworkService implements Runnable {
   private final ServerSocket serverSocket;
   private final ExecutorService pool;

   public NetworkService(int port, int poolSize) throws IOException {
     serverSocket = new ServerSocket(port);
     pool = Executors.newFixedThreadPool(poolSize);
   }

   public void run() { // run the service
        try {
           for (;;) {
             pool.execute(new Protocol(serverSocket.accept()));
           }
       } catch (IOException ex) {
            pool.shutdown();
        }
     }
 }
```



Executor

```
class Protocol implements Runnable {
   private final Socket socket;
   Protocol(Socket socket) { this.socket = socket; }
   public void run() {
     // read and service request on socket
   }
 }
```