Threads

- A thread is a flow of control in a program.
- The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.
- When a Java Virtual Machine starts up, there is usually a single thread (which typically calls the method named main of some designated class).
- Threads are given priorities. A high priority thread has preference over a low priority thread.

Understanding Threads

- You must be able to answer the following questions
  - What code does a thread execute?
  - What states can a thread be in?
  - How does a thread change its state?
  - How does synchronization work?

Thread Objects

- As is everything else, threads in Java are represented as objects.
- The code that a thread executes is contained in its `run()` method.
  - There is nothing special about run, anyone can call it.
- To make a thread eligible for running you call its `start()` method

Example

```
public class CounterThread extends Thread {
  public void run() {
    for ( int i=0; i<10; i++)
      System.out.println("Count:  " + i);
  }

  public static void main(String args[]) {
    CounterThread ct = new CounterThread();
    ct.start();
  }
}
```


Interface Runnable

- Classes that implement `Runnable` can also be run as separate threads
- `Runnable` classes have a `run()` method
- In this case you create a thread specifying the `Runnable` object as the constructor argument

Example

```
public class DownCounter implements Runnable {
  public void run() {
    for (int i=10; i>0; i--)
      System.out.println("Down:  "+ i);
  }

  public static void main(String args[]) {
    DownCounter ct = new DownCounter();
    Thread t = new Thread(ct);

    t.start();
  }
}
```

Many

```
public class Many extends Thread {
  private int retry; private String info;

  public Many (int retry, String info) {
    this.retry = retry; this.info = info;
  }

  public void run () {
    for (int n = 0; n < retry; ++ n) work();
      quit();
  }

  protected void work () { System.out.print(info); }
  protected void quit () { System.out.print('\n'); }

  public static void main (String args []) {
    if (args != null)
      for (int n = 0; n < args.length; ++n)
        new Many( args.length, args[n]).start();
}}
```

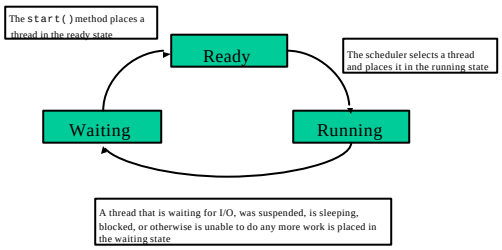

When Execution Ends

- The Java Virtual Machine continues to execute threads until either of the following occurs:
  - The exit method of class `Runtime` has been called
  - All threads that are not *daemon* threads have died, either by returning from the call to the `run()` or by throwing an exception that propagates beyond `run()`.
- You cannot restart a dead thread, but you can access its state and behavior.

Thread Scheduling

- Threads are scheduled like processes
- Thread states
  - Running
  - Waiting, Sleeping, Suspended, Blocked
  - Ready
  - Dead
- When you invoke `start()` the Thread is marked ready and placed in the thread queue

Thread States



Scheduling Implementations

- Scheduling is typically either:
  - non-preemptive
  - preemptive
- Most Java implementations use preemptive scheduling.
  - the type of scheduler will depend on the JVM that you use.
  - In a non-preemptive scheduler a thread leaves the running state only when it is ready to do so.

Thread Priorities

- Threads can have priorities from 1 to 10 (10 is the highest)
- The default priority is 5
  - The constants Thread.MAX_PRIORITY, Thread.MIN_PRIORITY, and Thread.NORM_PRORITY give the actual values
- Priorities can be changed via `setPriority()` (there is also a `getPriority()`)

isAlive()

- The method `isAlive()` determines if a thread is considered to be alive
  - A thread is alive if it has been started and has not yet died.
- This method can be used to determine if a thread has actually been started and has not yet terminated

isAlive()

```
public class WorkerThread extends Thread {
  private int result = 0;

  public void run() {
    // Perform a complicated time consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    while ( t.isAlive() );  // What is wrong with this?

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


sleep()

- Puts the currently executing thread to sleep for the specified number of milliseconds
  - `sleep(int milliseconds)`
  - `sleep(int millisecs, int nanosecs)`
- Sleep can throw an `InterruptedException`
- The method is static and can be accessed through the `Thread` class name

sleep()

```
public class WorkerThread extends Thread {
  private int result = 0;

  public void run() {
    // Perform a complicated time consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    while ( t.isAlive() )
      try {
        sleep( 100 );
      } catch ( InterruptedException ex ) {}

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


Timer

```
import java.util.Date;

class Timer implements Runnable {
  public void run() {
    while ( true ) {
      System.out.println( new Date() );

      try {
        Thread.currentThread().sleep(1000);
      }
      catch ( InterruptedException e ) {}
    }
  }

  public static void main( String args[] ) {
    Thread t = new Thread( new Timer() );

    t.start();
    System.out.println( "Main done" );
  }
}
```


yield()

- A call to the `yield()` method causes the currently executing thread to go to the ready state (this is done by the thread itself)

yield()

```
public class WorkerThread extends Thread {
  private int result = 0;

  public void run() {
    // Perform a complicated time consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    while ( t.isAlive() )
      yield()

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


Joining Threads

- Calling `isAlive()` to determine when a thread has terminated is probably not the best way to accomplish this
- What would be better is to have a method that once invoked would wait until a specified thread has terminated
- `join()` does exactly that
  - `join()`
  - `join(long timeout)`
  - `join(long timeout, int nanos)`
- Like `sleep()`, `join()` is static and can throw an `InterruptedException`

join()

```
public class WorkerThread extends Thread {
  private int result = 0;

  public void run() {
    // Perform a complicated time consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    try {
        t.join();
    } catch ( InterruptedException ex ) {}

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

Problems!!

```
import java.util.*;

public class Sync extends Thread {
    private static int common = 0;
    private int id;

    public Sync( int id ) { this.id = id; }

    public void run() {
        for ( int i = 0; i < 10; i++ ) {
            int tmp = common; tmp = tmp + 1;

            try {
                Thread.currentThread().sleep( 10 );
            } catch ( InterruptedException e ) {};

            common = tmp;
        }
    }
```


Problems!!

```
public static void main( String args[] ) {
    int numThreads = 0;
    try {
        numThreads = Integer.parseInt( args[ 0 ] );
    } catch ( NumberFormatException e ) { System.exit( 1 ); }

    List threads = new ArrayList();
    for ( int i = 0; i < numThreads; i++ ) {
        threads.add( new Sync( i ) );
        ( ( Thread )threads.get( i ) ).start(); }

    Iterator i = threads.iterator();
    while ( i.hasNext() )
        try {
          ( (Thread)i.next() ).join();
        } catch( InterruptedException e ) {};

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


Synchronization

- Every object has a *lock* that can be held by at most one thread at a time
  - A thread gets a lock by entering a synchronized block of code
- A thread can give up a lock by:
  - leaving a block of synchronized code
  - calling *lock*.`wait()`
- A thread executing `wait()` can be released by:
  - `notify()`
    - some waiting thread is allowed to compete for the lock
  - `notifyAll()`
    - all waiting threads are allowed to compete for the lock

Synchronized Code

- There are two ways to mark code as synchronized:
  - use the `synchronize` statement

```
synchronize( someObject) {
    // must obtain lock to enter this block.
    // wait()ing threads have to reacquire the
    //   lock before they are allowed to proceed.
}
```

  - using the synchronized method *shorthand*

```
public synchronized someMethod() { … }
```

  - which the same as

```
public someMethod() {
    synchronized( this ) { … } }
```


Example

```
import java.util.*;

public class Sync extends Thread {
    private static int common = 0;
    private int id;
    private Object lock;

    public Sync( int id, Object lock ) {
       this.id = id;  this.lock = lock;
    }

    public void run() {
       for ( int i = 0; i < 10; i++ )
           synchronized( lock ) {
              int tmp = common; tmp = tmp + 1; common =  tmp;
           }

       yield();
    }
```


Example

```
public static void main( String args[] ) {
   int numThreads = 0;
   try {
       numThreads = Integer.parseInt( args[ 0 ] );
   } catch ( NumberFormatException e ) { System.exit( 1 ); }

   List threads = new ArrayList();
   Object theLock = new Integer( 0 );

   for ( int i = 0; i < numThreads; i++ ) {
       threads.add( new SyncFixed( i, theLock ) );
       ( ( Thread )threads.get( i ) ).start();  }

   Iterator i = threads.iterator();
   while ( i.hasNext() )
       try { (Thread)i.next() ).join();
       } catch( InterruptedException e ) {};

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

Test 1

```
public class Locks1 extends Thread {
  private Object lock; private int myId;

  public Locks1( Object l, int id ) { lock = l; myId = id; }

  public void method() {
    synchronized( lock ) {
      for ( int i = 0; i < 3; i++ ) {
        System.out.println( "Thread #" + myId + " is tired" );
        try {
          Thread.currentThread().sleep( 10 );
        } catch ( InterruptedException e ){}
        System.out.println("Thread #" + myId + " is rested" ); }}}

  public void run() { method(); }

  public static void main( String args[] ) {
      Integer lock = new Integer( 0 );
      for ( int i = 0; i < 3; i++ ) new Locks1( lock, i ).start(); }}
```


Answer 1

Since all the threads are using the same object for the lock, each thread will run its `method()` to completion before another thread can get the lock.

```
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
```


Test 2

```
public class Locks2 extends Thread {
  private Object lock = new Integer( 0 ); private int myId;

  public Locks2( int id ) { myId = id; }

  public void method() {
    synchronized ( lock ) {
      for ( int i = 0; i < 3; i++ ) {
        System.out.println( "Thread #" + myId + " is tired" );
        try {
          Thread.currentThread().sleep( 10 );
        } catch ( InterruptedException e ){}
        System.out.println("Thread #" + myId + " is rested" );
    }}}

  public void run() { method(); }

  public static void main( String args[] ) {
    for ( int i = 0; i < 3; i++ ) new Locks2( i ).start(); }}
```

Answer 2

There is no synchronization here because each thread has a different lock. the thread still has to get the lock to enter the synchronized block, but since the lock s are all different the synchronization is lost.

```
Thread #1 is tired
Thread #2 is tired
Thread #0 is tired
Thread #1 is rested
Thread #1 is tired
Thread #2 is rested
Thread #2 is tired
Thread #0 is rested
Thread #0 is tired
Thread #1 is rested
Thread #1 is tired
Thread #2 is rested
Thread #2 is tired
Thread #0 is rested
Thread #0 is tired
Thread #1 is rested
Thread #2 is rested
Thread #0 is rested
```


Test 3

```
public class Locks3 extends Thread {
  private static Object lock = new Integer( 0 ); private int myId;

  public Locks3( int id ) { myId = id; }

  public void method() {
    synchronized ( lock ) {
      for ( int i = 0; i < 3; i++ ) {
        System.out.println( "Thread #" + myId + " is tired" );
        try {
          Thread.currentThread().sleep( 10 );
        } catch ( InterruptedException e ){}
        System.out.println("Thread #" + myId + " is rested" );
  }}}

  public void run() { method(); }

  public static void main( String args[] ) {
    for ( int i = 0; i < 3; i++ ) new Locks3( i ).start(); }}
```


Answer 3

Here we have synchronization because the lock is a static member . This means that regardless of the number of objects that are instantiated from this class, there w ill always be exactly one lock.

```
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
```

Test 4

```
public class Locks4 extends Thread {
  private int myId;

  public Locks4( int id ) { myId = id; }

  public synchronized void method() {
    for ( int i = 0; i < 3; i++ ) {
      System.out.println( "Thread #" + myId + " is tired" );
      try {
        Thread.currentThread().sleep( 10 );
      } catch ( InterruptedException e ){}
      System.out.println("Thread #" + myId + " is rested" );
    }
  }

  public void run() { method(); }

  public static void main( String args[] ) {
    for ( int i = 0; i < 3; i++ ) new Locks( i ).start(); }}
```


Answer 4

No synchronization because each thread is locking on a different `Locks4` object.

```
Thread #0 is tired
Thread #1 is tired
Thread #2 is tired
Thread #0 is rested
Thread #0 is tired
Thread #1 is rested
Thread #1 is tired
Thread #2 is rested
Thread #2 is tired
Thread #0 is rested
Thread #0 is tired
Thread #1 is rested
Thread #1 is tired
Thread #2 is rested
Thread #2 is tired
Thread #0 is rested
Thread #1 is rested
Thread #2 is rested
```


SyncQueue

```
public class SyncQueue {
   private Object q[]; private int head; private int tail;
   private int count; private int cap;

   public SyncQueue(int size) {
     q = new Object[size]; head = 1; tail = 0; count = 0; cap = size; }

   public synchronized void enqueue(Object o) {
     if ( !isFull() ) { tail = ( tail + 1 ) % cap; q[ tail ] = o; count++; }}

   public synchronized Object dequeue() {
     Object retval = null;
     if ( !isEmpty() ) { retval = q[ head ]; head = ( head + 1 ) % cap; count--; }
     return retval; }

   public Object peek() {
     Object retval = null;
     if ( !isEmpty() ) retval = q[head];
     return retval;}

   public boolean isEmpty() { return count == 0; }
   public boolean isFull() { return count == cap; }}
```

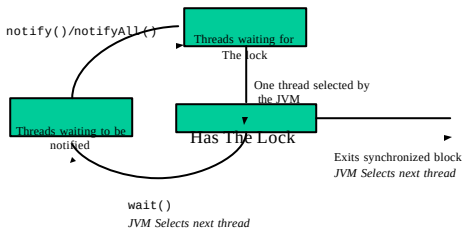
Synchronized Static Methods

- Java also provides synchronized static methods.
- Before a synchronized static method is executed, the calling thread must first obtain the class lock.
- Since there is only one class lock, at most one thread can hold the lock for the class (object locks can be held by different threads locking on different instances of the class).

`wait()/notify()`

- In all of the previous examples a thread gave up a lock when it left the synchronized block
- It is possible for a thread to give up a lock while it is in a synchronized block
  - The method `wait()` is executed on the object whose lock the thread is holding
- The thread will resume execution via a call to the lock object's `notify()` method

`wait()/notify()`



Customer

```
public class Customer extends Thread {
    public static int MAX_ITEMS = 25; // Max items
    private int id;                   // This customers id
    private int numItems;             // The numebr of items for this customer
    private Cashier register;         // The only register in the store

    public Customer( int id, Cashier register ) {
        this.id = id;
        this.register = register;

        numItems = (int)( Math.random() * MAX_ITEMS ) + 1;
    }

    public void run() {
        register.checkOut();
        System.out.println( "Customer " + id + " is checking out" );

        try { sleep( 500 ); } catch ( InterruptedException e ) {}

        System.out.println( "Customer " + id + " is leaving the line" );
        register.done();
    }
}
```


Cashier

```
public interface Cashier {

    /**
     * Invoked by a customer when they are ready to check out.
     * Contains the logic required to select the one customer
     * that the cashier will service.  If the cashier is already
     * serving another customer, this customer will wait until
     * the other customer has finished with the cashier.
     */

    public void checkOut();

    /**
     * Invoked by a customer when they are finished with the
     * cashier.  If customers are waiting for the cashier,
     * one of the waiting customers will be selected and
     * serviced by the cashier.
     */

    public void done();

} // Cashier
```


Cashier1

```
public class Cashier1 implements Cashier {
    private boolean busy = false;  // Is the cashier busy?

    public synchronized void checkOut() {
        // While the cashier is busy -- wait

        while ( busy )
            try {
                wait();
            } catch ( InterruptedException e ){}

        busy = true;
    }

    public synchronized void done() {
        if ( busy ) {
            busy = false;
            notifyAll();
        }
    }

} // Cashier1
```

Cashier2

```
import java.util.*;

public class Cashier2 implements Cashier {
    private boolean busy = false;  // Is the cashier busy?
    private int tenOrLess = 0;     // How many folks have 10 or fewer

    public synchronized void checkOut() {
        Customer me = (Customer)Thread.currentThread();
        int items = me.getNumItems();

        if ( items <= 10 ) tenOrLess++;
        while ( busy || tenOrLess > 0 && items > 10 )
            try { wait(); } catch (InterruptedException e ){}
        busy = true;
    }

    public synchronized void done() {
        if ( busy ) {
            Customer me = (Customer)Thread.currentThread();
            if ( me.getNumItems() <= 10 ) tenOrLess--;
            busy = false;
            notifyAll();
        }}
```


suspend()

- The thread class has a `suspend()` method
  - A suspended thread remains suspended until it is resumed by another thread
- This method has been deprecated, as it is deadlock-prone:
  - A suspended thread holds any locks that it may have.
  - If the thread holds a lock, no thread can obtain the lock until the target thread is resumed.
- If the thread that would resume the target thread attempts to obtain the lock prior to calling resume, deadlock results.
- `resume()` is deprecated as well (it is more or less useless without `suspend()`

stop()

- The thread class has a `stop()` method
  - Forces a thread to stop executing
  - Can be invoked by another thread
- This method has been deprecated, as it is unsafe:
  - Stopping a thread with Thread.stop causes it to release all of the locks that it has holding
  - If any of the objects protected by these locks are in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior.
- Many uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running.

Example

```
private boolean continue = true;

public void stop() {
  continue = false
}

public void run() {
  while (continue) {
    try {
      thisThread.sleep(100);
    } catch ( InterruptedException e){
  }
}
```