Multi-Threading

Agenda

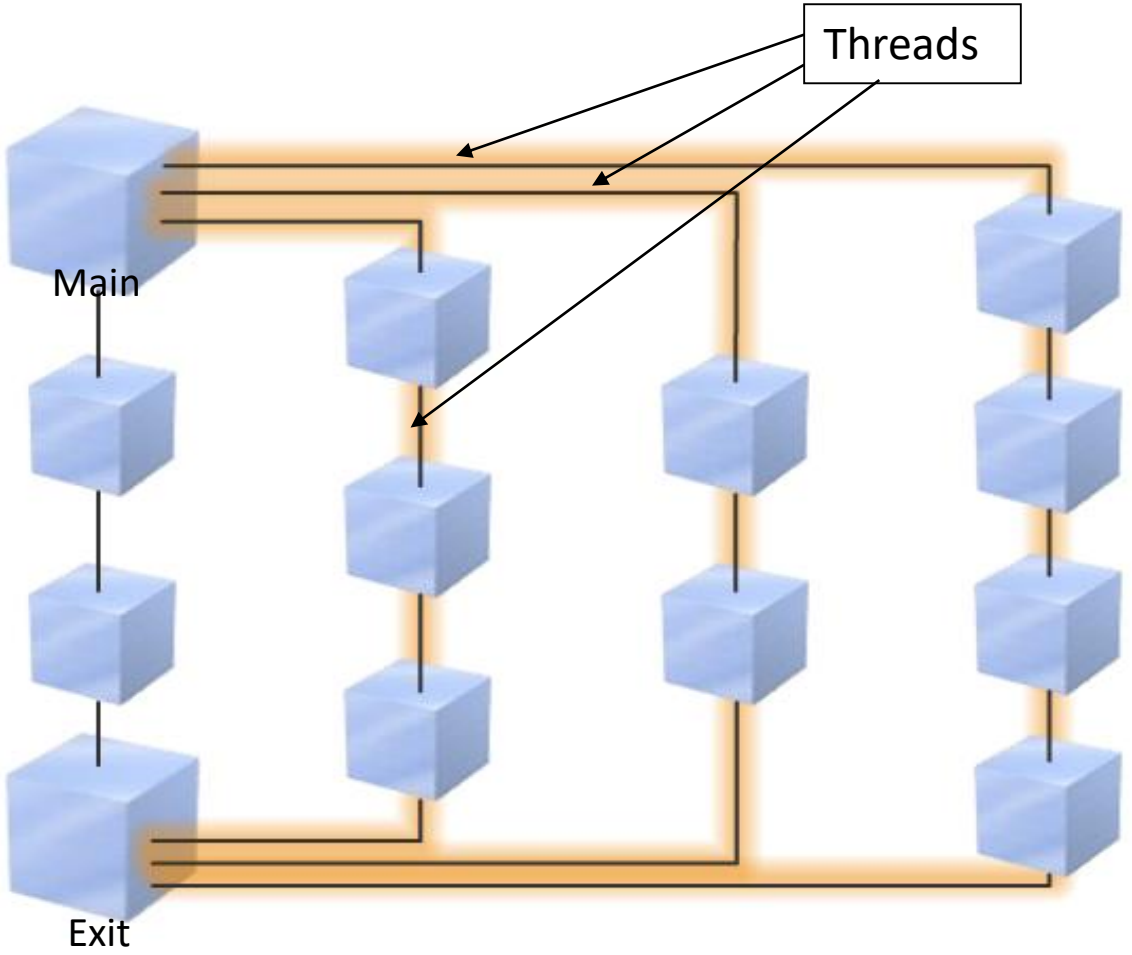
- What is Thread?
- Benefits of Multi-Threading
- Downsides of Multi-Threading
- Defining and Starting a Thread
- Thread Priorities
- Sleeping, Waiting, Yielding, Joins, Interrupts, etc
- Naming Threads
- Thread States
- Daemon Threads
- Synchronization
- wait/notify/notifyAll

Definitions

- Concurrency
  - A software that can do more than one thing at a time is known as concurrent software.
- Process
  - A process is a self-contained running program with its own address space. A multitasking operating system is capable of running more than one process (program) at a time.
  - Most implementations of the Java virtual machine run as a single process.
  - A Java application can create additional processes using a ProcessBuilder object. (out of scope of this course)
- Thread
  - Threads exist within a process — every process has at least one
  - A thread is a single sequential flow of control within a process. A single process can thus have multiple concurrently executing threads.

What is Thread?

Sequential - Single Thread



Multi-Threaded

Benefits of Multi-Threading

- Multiple threats are created in order to:
  - Allow a program to do more than one thing at a time.
  - Maximize the use of the CPU
    - Some sorting and merging algorithms, some matrix operations, and many recursive algorithms
  - Maximize the productivity of the users
    - While the user entering data (a task with a very low CPU utilization), the CPU can be used to calculate some information. Data entry and calculation should be running in separate threads.
    - Trying to load a large file and display its content to the user versus using multiple threads, reading the file in one thread while displaying the file in another
  - Run some task at the same time as another task
    - The classic example is the server part of a client/server. When each request comes in from a client, it is very convenient if the server can create a new thread to process that one request.
  - Take advantage of multiple processors of some machines
  - Make an application appear more responsive
    - interactive programs that never "go dead" on the user: one thread controlling and responding to a GUI, another thread carries out the tasks or computations requested, third thread does file I/O, all for the same program.

Downsides of Multi-Threading

- ***Creating a thread takes CPU cycles***. Simple calling a function may be faster. Therefore, it is best just to call a small, short-lived function rather than to start it as a separate thread.
- Running multiple threads requires the CPU to juggle more than one thing at a time. ***This juggling takes time***. If you had 100,000 threads running at the same time, the CPU would spend all of its time managing to switch between one thread to another (context switching)
- Multi-threaded applications can be ***more complex to write***. You have to be sure that two threads are not accessing the same resource at the same time. Imagine one thread writing to an object while second one was reading from that object.
- If you are not careful, multi-threaded programs might appear to hang. ***Deadlock issue***.
- Multi-threaded applications can be ***very difficult to debug***. It might be very hard to reproduce this type of problems.
- Multi-threading must support these mechanisms
  - Synchronize with each other
  - Prevent simultaneous access to specific pieces of data
  - Wait for each other to complete various tasks
- BOTTOM LINE USE THEM ONLY IT IS NECESSARY!

Thread Fundamentals

- One of the great thing about Java is that it has built-in support for writing multi-threaded programming since its beginning
- In any JVM you have at least two threads created for you. One is the working thread and the other one is the Garbage Collector thread
- Thread supports can be found in:
  - The java.lang.Object class
    - notify(), notifyAll(), wait(), wait(long timeout), wait(long timeout, int nanos)
  - The java.lang.Thread class
  - The java.lang.Runnable interface
  - Virtual Machine

Defining and Starting a Thread

- An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:
  - Provide a ***Runnable*** object. The Runnable interface defines a single method, ***run***, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the `HelloRunnable` example:
  - Subclass of ***Thread***. The Thread class itself implements Runnable, though its ***run*** method does nothing. An application can subclass Thread, providing its own implementation of run, as in the `HelloThread` example:

- In order to start a thread, we must instantiate the class and call the object's start method.
- Calling a start() method will ***register*** the thread with a piece of system code called ***thread scheduler***
- Thread scheduler might be part of the JVM or part of the OS.
- The scheduler determines which thread is running on each CPU on any given time.
- We never call the ***run*** method directly. It is called as a result of our starting the Thread

How Does a Thread Execute?

- The Thread executes its own run() method
  - If we want the thread to execute its own run () method, we should subclass the Thread and define a run() method in the class.
  - We must override run() method. The default implementation of class thread does nothing.
  - To make this code run we should construct and instance of the Class and invokes its start() method.

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

- The Thread executes another classes run() method
  - Implement Runnable interface
  - We must implement run() method of Runnable interface
  - Invoke alternate constructor from Thread class
    - Public Thread (Runnable target)

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

When Does a Thread End?

- When ***run()*** method returns or ends
  - When the run method ends, it will cause the thread to terminate naturally.
  - If you want the thread task to be performed again you have to construct and start a new thread
  - The dead thread continues to exist as an object in the memory and you still can access it but you can't start it again.
- In case of and Exception
  - This is exactly like run() returns but it means that the thread fails to perform all of its tasks.
- Declaring ***volatile*** type of variable within the same class
  - Compiler always checks value of ***done*** variable at the start of each while iteration
  - ***volatile*** keyword prevents optimization
  - Preferred way

```
private volatile boolean done=false;
public void run() {
   while (!done) {
      System.out.println(".");
   }
}
public void kill() {
   done = true;
}
```

start() and isAlive()

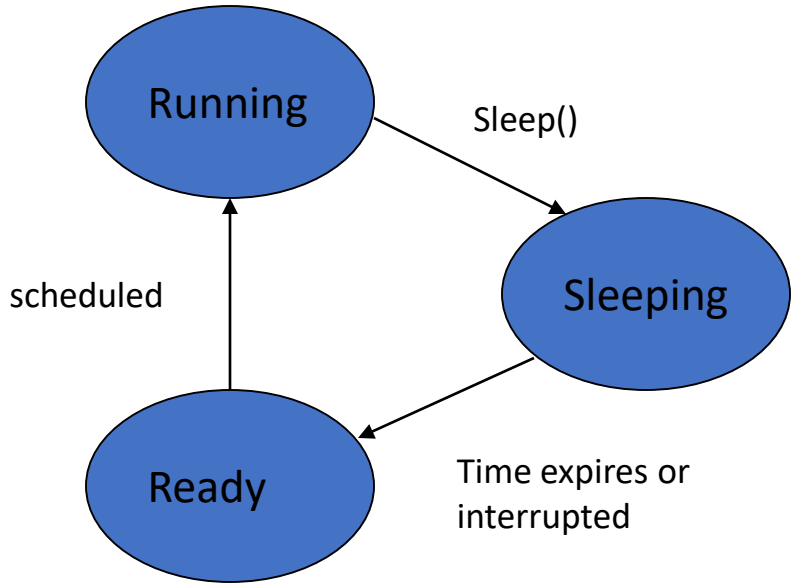
- isAlive() methods tells you if the thread is alive or not
- Example
  - `AliveTwoThread`

Thread Priorities

- Each thread has a priority
- The priority is an integer from 1 to 10
- The default priority is 5
- Thread class has three constants.
  - `MIN_PRIORITY = 1;`
  - `NORM_PRIORITY = 5;`
  - `MAX_PRIORITY = 10;`
- Threads with higher priority gets the next processor over those with low priority if all other things being equal
- The priority is considered by thread scheduler when it decides which ready thread should execute
- If there is more than one thread waiting, the scheduler chooses one of them. However, there is no guarantee that the thread chosen will be the one that has been waiting the longest
- To set the thread priority call setPriority () method by passing the desired value
- The getPriority () returns priority. It is used with currentThread() method
- Example
  - `SetPriority`

Pausing Execution with Sleep

- Thread.sleep causes the current thread to suspend execution for a specified period
- That means thread is alive but not scheduled to use any CPU cycles.
- Any thread can be put to sleep except the threads used in UI code
- Since the method sleep() is a static method of the standard Java runtime class Thread, we can just invoke it with the class name.
  - Thread.sleep(long millis)
  - Thread.sleep(long millis, int nanos)
- Example
  - `SetPriority`



Interrupts

- A sleeping thread can be interrupted
- The interrupt() method of the Thread class breaks a thread from sleep
- When a Thread is interrupted from sleeping, the sleep function throws the interruptedException
- isInterrupted() is a way for a thread to detect if it has been interrupted. Remember the interruptedException will not be thrown unless the thread is sleeping
- The interrupt() method of Thread sets the interrupt state (a boolean flag) of a thread to "true.“
- Example
  - `PiInterrupt`

Joins

- We use join where the main thread must wait for the worker threads to complete before the main thread can continue
- The join method allows one thread to wait for the completion of another.
- Like sleep, join responds to an interrupt by exiting with an InterruptedException
- One thread may call join( ) on another thread to wait for the second thread to complete before proceeding.
- If a thread calls t.join( ) on another thread t, then the calling thread is suspended until the target thread t finishes
- The call to join( ) may be aborted by calling interrupt( ) on the calling thread, so a try-catch clause is required.
- Three overloaded versions
  - `public final void join()`
  - `public final synchronized void join(long millis)`
  - `public final synchronized void join(long millis, int nanos)`
- Example
  - `JoinExample2`

Naming Threads

- Threads can be named using setName() method
- getName is used to read name of Thread
- Good practices about naming a thread
  - Invoke setName() on the Thread before start.
  - Do not rename the thread after it is started
  - Give each thread a brief, unique and meaningful name
  - Do not change the names of JVM threads such as main
- Example
  - `SimpleThreadsWithName`

Yield – Cooperative Threads

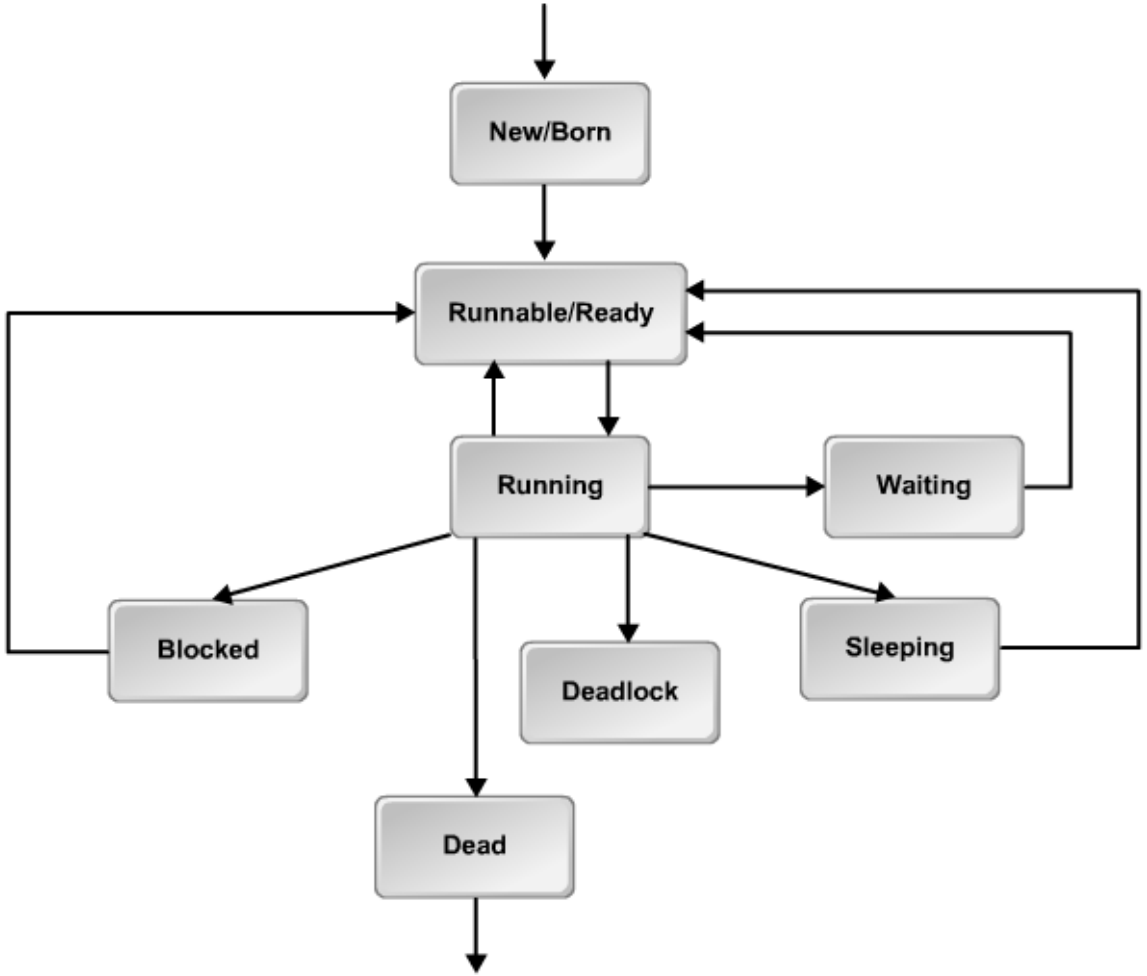
- Applications should be developed so that they don't hog the machine
- Threads should be written so that they cooperative with the JVM
- Periodically, your Threads should give up the CPU and let some other thread have it for a while
- To do this, we use either sleep() or yield()
- Example-1
  - In this thread, after every 1000 iterations of a loop, the processor is returned to the JVM.

```
public class Cooperative extends Thread {
    public void run() {
        while (true) {
            for (int i=0; i<10000000; i++) {
            if (i % 1000) yield();
            }
        }
    }
}
```

- Example-2

```
YieldingThread
```

Thread States



Thread States

Thread State	Description
New/Born	After the thread is instantiated, the thread is in the New state until the start() method is invoked. In this state, the thread is not considered alive and it is not using CPU
Runnable/Read	A thread comes into the runnable state when the start() method is invoked on it. It can also enter the runnable state from the running state or blocked state. The thread is considered alive when it is in this state.
Running	A thread moves from the runnable state into the running state when the thread scheduler chooses it to be the currently running thread. At this state, the thread is executing code in the run method.
Dead	A thread is considered dead when its run() method is completely executed or stop() method is invoked on the thread. A dead thread can never enter any other state, not even if the start() method is invoked on it.
Blocked	A thread may enter a blocked state while waiting for a resource like I/O or the lock of another object. In this case, the thread moves into the runnable state when the resource becomes available.
DeadLock	This states occurs when two threads are waiting for each other to complete something. That means, both threads can wait indefinetely. Example: DeadLockExample
Waiting	A thread is put into a waiting state by calling the wait() method. A call to notify() or notifyAll() may bring the thread from the waiting state into the runnable state. The sleep() method puts the thread into a sleeping state for a specified amount of time in miliseconds
Sleeping	Thread goes to sleep with the sleep() method

Deamon Threads

- A Java program exits when all of its threads have completed, but this is not exactly correct. What about the hidden system threads, such as the garbage collection thread and others created by the JVM? We have no way of stopping these. If those threads are running, how does any Java program ever exit?
- These system threads are called daemon threads. A Java program actually exits when all its non-daemon threads have completed.
- Any thread can become a daemon thread. You can indicate a thread is a daemon thread by calling the Thread.setDaemon(boolean on) method.
- Thread.getDeamon() is used to retrieve the deamon status of a thread
- You might want to use daemon threads for background threads that you create in your programs, such as timer threads or other deferred event threads, which are only useful while there are other non-daemon threads running.
- When an application ends, it will not terminate until all non-deamon threads have terminated. A deamon thread is useful when you want some tasks running, but do not want that task to stop the application from terminating. In another words, if a thread dies and there are no other threads except daemon threads alive, the Java virtual machine stops.
- Example: `SimpleDaemons`

Synchronization

- Synchronization allows us to ensure that threads see consistent views of memory.
- It ensures that only one thread executes a protected section of code at one time (mutual exclusion), and it ensures that data changed by one thread is visible to other threads (visibility of changes).
- It is possible for two threads executing on two different processors to see two different values for the same variable! This sounds scary, but it is normal. It just means that you have to follow some rules when accessing data used or modified by other threads.
- Without proper synchronization, it is possible for threads to see stale values of variables or experience other forms of data corruption.
- The ***synchronized*** keyword can be applied to a class, method, or block of code.
- Example: `week12.sync_lock.Consumer`
- Example: `week12.SafeCollectionIterationTest`

Synchronization

- Synchronized Methods
  - If the methods of an object should only be executed by one thread at a time, then the definitions of all such methods should be modified with the keyword ***synchronized*** .
  - Note that in this case the object whose lock will provide the mutual exclusion is implicit. It is the "this" object on which the method is invoked.
  - Synchronized methods are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently.

```
synchronized void method_name() { ...  }
```

This is equivalent to:

```
void method_name() {
  sychronized (this) {
      ...
  }
}
```

- Example: `week12.ObjectLock`

Synchronization

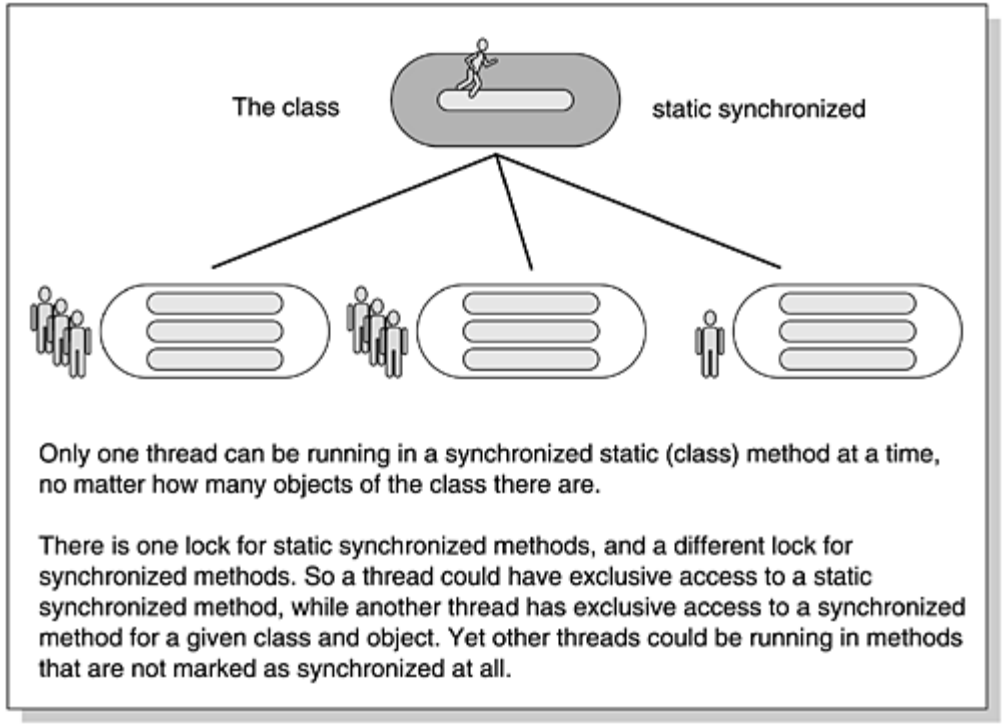
- Synchronized Block.
  - This is achieved by attaching the keyword " synchronized " before a block of code. You also have to explicitly mention in parentheses the object whose lock must be acquired before the region can be entered.
  - The code block is usually related to the object on which the synchronization is being done.

```
class SmartClient{
  BankAccount account;
  //…
  public void updateTransaction() {
    synchronized (account) {
            account.update();
        }
  }
}
```

- Tips: The basic rule for synchronizing for visibility is that you must synchronize whenever you are:
  - Reading a variable that may have been last written by another thread
  - Writing a variable that may be read next by another thread

Synchronization

- Mutual Exclusion Over an Entire Class.
  - This is achieved by applying the keyword synchronized to a class method (a method with the keyword static).
  - Only one static synchronized method for a particular class can be running at any given time, regardless of how many objects there are. The threads are implicitly synchronized using the Class object.
  - Synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.



wait/notify/notifyAll

- The Object class defines the methods wait(), notify(), and notifyAll(). To execute any of these methods, you must be holding the lock for the associated object.
- Wait puts a thread to sleep until a notify occurs
- It tells the OS that this thread should not receive any CPU cycles until the item being waited upon is available
- Wait() causes the calling thread to sleep until it is interrupted with Thread.interrupt(), the specified timeout elapses, or another thread wakes it up with notify() or notifyAll().
- When notify() is invoked on an object, if there are any threads waiting on that object via wait(), then one thread will be awakened. When notifyAll() is invoked on an object, all threads waiting on that object will be awakened.
- In particular, using notify() instead of notifyAll() is risky. Use notifyAll() unless you really know what you're doing.
- Example: `Producer-Consumer Mailbox`

Debugging Multi-Threaded Programs

- Debugging multi-threaded programs can be one of the toughest challenges that a developer can face. The problems are often related to synchronization of the threads. These situations are often very hard to reproduce.
- Tips
  - Never assume that one thread will complete or start before another, unless you put specific ***waits*** in place to make it happen
  - Never assume that the processor will interrupt a thread to start another one
  - Do not assume consistency
  - Never assume that the machine can only execute one instruction at a time. For example, NT, Unix and other Os's support multiple processors. Each processor can be executing a different thread at the same time
  - Avoid static variables within threads. These make threads non re-entrant and so static variables are a poor choice for multi-threading programming

Additional Reading and Other Resources

- ***Introduction to Java Threads by IBM developerWorks***
  - ***https://www6.software.ibm.com/developerworks/education/j-threads/j-threads-a4.pdf***
- ***Concurreny Tutorial by Oracle***
  - ***http://docs.oracle.com/javase/tutorial/essential/concurrency/***
- ***Just Java 2 by Peter van der Linden***
  - ***Chapter 10. Doing Several Things at Once: Threads***
  - ***Chapter 11. Advanced Thread Topics***
- ***Thinking in Java 3rd Edition by Bruce Eckel***
  - ***Chapter 13: Concurrency***