* thread = generic term for program execution abstraction
  + the fundamental unit of processing that can be scheudled by an operating system
* multithreading
  + a program running two or more threads concurrently
    - separate points of execution
    - separate call stacks
    - shared memory between different threads
      * objects can potentially be seen by two or more threads
* in the beginning, the first computers ran one program at a time
  + but then computers were getting faster/cheaper gave rise to interactive computing with GUI's
    - multithreading gives the illusion of simulatenous progress
      * but in reality, the processor is dividing its time
        + it's that it happens so fast that it appears simultaneous
* threads extend OS mechanisms for process-level time division to within a program
  + program needs/wants to make progress on two or more tasks
    - in our case, we want two threads that are part of the same program to share memory and coordinate progress
* Ex:
  + class called LongRunner which counts from 0 to 1 million, which slows down the program
    - this class means that the UI requires multithreaded programming
* Multicore increases performance only if we can find ways to parallelize our tasks
  + sometimes this is very easy
    - no data or logic dependencies
    - aggregating data operations
  + sometimes this is very hard
* two basic mechanisms for threads:
  + extend Thread class
    - override run()
      * within run(), do the task that you want to perform on the new thread
    - call start()
      * creates new thread
      * executes run() on that thread

or

* implement Runnable interface
  + means it has a run() method
  + provide reference to"runnable" object as parameter of Thread constructor
  + call start() on the thread object
  + Ex:

public void actionPerformed(ActionEvent e){

(new Thread(this)).start();

}

* Mulit threaded access to memory can be hazardous
  + operation ordering
    - may need all threads to see changes to memory in a consistent way
      * actual order may or may not matter as long as all threads see the same effective order
    - exclusive access
      * operation ay involve many steps which must be done with exclusive access to memory
        + other threads must be prevented from accessing memory while operation in progress
    - Ex:

in a Bank account, if you withdraw and deposit at the same time, both actions don't know they're reading the same initial balance

thus, the withdraw thinks there's less money in the account than the deposit method after each is finished executing

* Thus, the account object **is not thread safe**
* Thread.sleep(2000);
  + static method of Thread that allows you to take current thread of control to sleep for a certain number of milliseconds
* Object Monitor
  + Every object in Java is assoicated with a "monitor"
    - provides ability to "lock" the object
    - only one thread can "own" the monitor at a time
    - strategy is to lock object before executing any unsafe code
      * thus if one thread is already using the object, if another wants to use the object it is told to wait
      * as a programmer you just define which objects can never be handled by two threads at the same time
  + two ways to acquire the monitor lock:
    - synchronized methods
    - synchronized statements
* Synchronized methods
  + to create one, just add "synchronized" keyword after the public/private/protected

Ex: public synchronized void some\_method(){....}

* only one thread can be executing this method at one time
* all synchronized methods must obtain monitor lock of object before execution
  + automatically released when method returns
  + this means that for all synchronized methods of an object, only one can be executing at any given time
* Synchronized statements
  + Syntax:

synchronized(object) {...}

* specifies a block of code that must be only in one thread
* lock is obtained on monitor associated with object before statement block is executed
* useful if bits of code that you're trying to lock are spread everywhere
  + whereas synchronized methods, the object's lock you're using is the object that contains the method
  + with synchronized statements, this isn't necesarily the case
* a Synchronized method is essentially calling a synchronized statement, passing in this into the synchronized parenthesis

Deadlock

* occurs when a synchronnized method in object A calls a synchronized method in object B
  + obtained lock on A
  + waiting for lock on B
  + causes a deadlock, stuck forever because methods don't know that waiting won't fix their problems

Join

* sometimes one thread needs to wait for another thread to be done before continuing
  + a common form of thread coordination
* join()
  + method provided by Thread
  + when called in a different thread, the calling thread is blocked until called thread is done
    - the thread that calls the join() waits for the second thread to be finished before it can do anything
    - Done = run() method has finished
    - blocked means join() does not return until called thread is done(thus calling thread is waiting)