**Threads**

* “Threads allow a program to operate more efficiently by **doing multiple processes at the same time**.” (Concurrently!) They can be used to perform longer, more complicated tasks in the background **without interrupting the main program.**
  + You can think of a thread like a “process” that your computer/the JVM runs. Any program you run is technically a thread.
* There are **two ways to create a thread in Java**: Extend the **Thread Class** or Implement the **Runnable Interface.**
  + A Class can extend the Thread Class and override the run() method.
  + A Class can Implement Runnable, which will also require you to implement the run() method.
* Then, **calling the** **start() method will create a thread** and then use whatever is in that run() method as the base method for execution flow.
  + So you can invoke start() 5 times, and 5 threads would start.
* **States of a thread**
  + **New:** newly created thread that has not started executing.
  + **Runnable:** Thread is either running or ready for execution but waiting for resource (memory) allocation (see below).
  + **Blocked:** Waiting to enter or re-enter the block or method from which it’s being called, usually due to no resources available for running.
  + **Waiting:** Waiting for some other thread to perform an action without any time limit.
  + **Timed\_Waiting:** Waiting for some other thread to perform an action with a specific time limit
  + **Terminated:** has completed its execution

\*HelloThreads demo\*

**The Producer-Consumer Problem**

* **Threads run independently**. What this means is that **there is no way to ensure the order in which they will run in comparison to each other.** This “multithreaded environment” has specific issues that can crop up because of this.
  + Imagine we have some fixed-size Queue that is constantly adding and removing elements.
    - A thread could try to add resources to a queue that’s full
      * producer failure
    - Or a thread could try to read or remove from a queue that’s empty
      * consumer failure
    - These are examples of **concurrency issues**
* Solving the producer consumer problem
  + We can solve the Producer-Consumer problem by using **wait()** & **notify()** methods to communicate between producer and consumer threads.
    - The **wait()** method can pause the producer or consumer thread depending on the queue size, typically when the queue is full.
    - The **notify()** method sends a notification to the waiting thread, typically once the queue is ready to be added to.
* **Some other concurrency issues:**  
  + **Deadlock:** occurs when two threads are **competing for the same set of resources,** and neither is willing to give up the resources they control. This can cause your application to stall and run indefinitely. This can usually be resolved by giving threads priority.
    - We have the setPriority() method, which we can attach to thread objects, it gets a value of 1-10 with 10 being the highest priority.
  + **Starvation:** is like deadlock in that threads are competing for resources. But in this case, **one thread has all the resources another thread needs.** The thread without resource access is stalled for a while. This can be mitigated by having threads release resources when they’re not actively using them, or just waiting for the thread to complete.
  + What are these “resources” the Threads are competing for?
    - Application Memory
    - CPU/Processor time
    - File handles and the data in those files
    - Database Connections (assuming there’s a finite amount)
* **Other ways to fix concurrency issues?** 
  + **Synchronization** - There is a non-access modifier “synchronized” which will “lock down” a method or class to a single thread having access to it at a time. When **we only want one thread to access the shared resource.**
    - **StringBuffer** is synchronized while StringBuilder is not. Hence why we use StringBuffer with Threads.
  + **yield()** is a good way to avoid deadlock. It will pause a thread, giving up its resources to the other thread that needs them. The thread will “yield” to other threads.
  + **join()** will combine the stack of the thread back with the main method (which is a thread!). The method will then wait for the thread to die. By “joining” the thread with the main method, it simply becomes part of the main method, thus, it gets taken off the top of the stack.