Threads

A thread is an execution on your computer. Usually your program is a single thread.

If you break your program into multiple threads, each thread will run independently and concurrently.

So each thread in your program is a piece of the program that can run independently and concurrently with the rest of your program.

For example, suppose you have a program that has a variable x that currently stores 1000. The program has 2 threads.

Thread 1: will perform x = x + 100;

Thread 2: will perform x = x + 500;

After both threads complete, the value stored in x should be 1600. But things can go wrong.

Consider the line x = x + 500. This is really 3 separate steps:

1. Get the value of x

2. Add 500 to that value.

3. Store the result back to x.

Now, suppose the two threads get interleaved as follows:

Thread 2: 1. Get the value of x. (Gets 1000.)

Thread 2: 2. Add 500 to that value. (Has value 1500.)

Thread 1: 1. Get the value of x. (Gets 1000.)

Thread 1: 2. Add 100 to that value. (Has value 1100.)

Thread 1: 3. Store the result back to x. (x now stores 1100.)

Thread 2: 3. Store the result back to x. (x now stores 1500.)

So after the threads complete, x stores 1500 instead of 1600!

The solution is to lock (the technical term is to use a "semaphore") certain parts of the code so that only one thread can manipulate variable x at a time.

Using Threads:

Java has a Thread class that lets us create a separate thread.

Thread implement the Runnable interface. So, you can create your own thread structures by implementing this interface.

The two most important methods of Thread:

run: This method is like the main method of your Java program. You override this method and put the code you want the thread to run in this method.

start: This method starts the execution of the thread.

See the BankAccount.java example for an example that creates 2 threads, one that adds money to a bank account and one that removes money from a bank account.

Some important Java keywords:

volatile: This informs Java that the value of the variables could be changed by another thread. We do this to doneDeposits and doneWithdraws because we have a loop:

while (!doneDeposits || !doneWithdraws)

and neither variable is changed in the loop body. The Java compiler, seeing this, might change the code so that it does not keep checking the value of the

variables, and the result will be an infinite loop. So, we put volatile on the variable declaration to let Java know that the value might change between

checks (eventhough this thread did not change the values) and so the program should still inspect the values with each loop iteration.

final: This indicates that the value of a variable will not change, once it has been initialized. We need to do this to the account variable because the anonymous

classes used to create the Threads are using this variable. Note that the account variable is a local variable in the method, it is allocated on the stack,

and thus the variable will be destroyed when the method exits. However, both threads will exist after the method exits, and need access to the variable.

If the threads try to modify the value of the variable after the method ends, we will have an error - accessing a variable after it has been deallocated.

By making the variable final, we indicate that is no chance that the value of the variable will change, and thus Java can copy the value of the variable to

the thread. With the value copied to the thread, the thread never needs to access the variable after the method exits.

synchronized: This is used to lock code. If you put synchronized on a method, then that method is locked. Any thread that wants to access a synchronized method

of the class can not if any other thread is in some synchronized method of the class. The thread will have to wait until the other threads are done with

the synchronized methods.

We can also synchronize blocks of code. This is usually better because it is a finer grain of locking. Also, when we synchronize a block of code, we use an

object as the lock. If a thread wants to enter code synchronized with an object, it must wait as long as some other thread is executing code that is

synchronized with that same object.

Why could we not use Integer balance as the object to lock the code that modifies the balance? Recall that Integer is a wrapper class. Each time we do

balance += amount; the balance is converted to int, and the a -new- Integer instance is created to store the balance. When using synchronized with an Object, all code

locked by that object. However, when we keep changing the balance, we keep getting new objects, and so each block is locked by a different object!! We might as well have

no locking at all.