Multithreading

Parallel Processing

- Some programs can run faster if tasks can be run in parallel
  - Calculations on large sets of numbers if operations can be done in parallel
    - $(1+3)$*$(2+7)$: additions can be done in parallel, then add results
  - Animation work calculates positions in parallel

Processes and Threads

- Process
    - A self contained execution environment
        - Every process has its own memory, cpu time, etc
    - Processes cannot share memory between themselves directly
- Thread
    - A program unit that runs independently of other parts of the program
    - Can have multiple threads in a single process
    - Can share memory with other threads in the same process

Threads

- Usually, the JVM executes each thread for a short period of time, then switches
    - This appears to run in parallel, but isn't
    - This works well; IO and network ops don't require the CPU
- Multiple processor/core computers allow multiple threads to run simultaneously

Threads in Java

- To run a new thread in Java
  - In the class that represents your thread
    - define a class that implements the `Runnable` interface
    - override the `run` method of the interface
  - In your driver
    - instantiate an object of your class
    - construct a `Thread` object from the object you instantiated
    - call the `start` method of the `Thread` object

Example

- See the included threading project for an example of basic threading

Terminating Threads

- A thread terminates when the `run` finishes
- To stop a thread manually, you should `interrupt` it
    - `stop` is deprecated, do not use it
    - When the `interrupt` method is called, a `boolean` field in the thread data structure is set
    - The `run` method can check for interruptions, do required cleanup, then exit

Dangers of Threading

- Race Conditions
  - Two threads compete for the same resources
  - Unpredictable results (who gets the resources first?)
  - Fixed using locks
- Locks not released
  - Fix using `finally`
- Deadlock
  - Fix using conditions

Race Condition Example

- Two threads are updating a bank acount (one is withdrawing $100, one is depositing $100)
  - initial balance is 0
  - thread 1 reads init7al balance of 0
  - thread 2 reads initial balance of 0
  - thread 1 deposits 100 and calculates (but doesn't store) the new balance of 100
  - thread 2 withdraws 100 and calculates (but doesn't store) the new balance of -100
  - thread 2 stores the new balance of -100
  - thread 1 stores the new balance of 100
  - final balance: 100! (could have been -100 if the thread order had been different)

Synchronization

- Lock objects can be used to solve race conditions
- `ReentrantLock` class
  - Use this with a class whose methods access shared resources
    ```
    balanceChangeLock = new ReentrantLock();
    ```
  - `lock` and `unlock` the `ReentrantLock` as required
    ```
    balanceChangeLock.lock();
    // code that uses shared resources
    balanceChangeLock.unlock();
    ```
  - Only one thread may have the object locked at a time; others must wait until it is unlocked again

Handling Exceptions

- If an exception is thrown between calls to `lock` and `unlock`, the call to `unlock` never happens
- To solve this, use a `try` block around the code using the shared resource, and place the call to `unlock` in a `finally` block

Deadlock

- We want to add a test to disallow negative balances
    - Add a test inside the try block: if the balance is less than the amount being withdrawn, we wait until the balance grows
- How do we wait?
    - If the thread sleeps, it will block other threads trying to acquire the same lock it's holding
    - Other threads will call deposit, but be blocked by the withdraw method
    - Withdraw can't complete until a deposit is made!

Condition Objects

- Conditions allow an object to temporarily release a loc and regain it later
- A condition belongs to a lock object
- The `newCondition` method allows a lock to obtain a condition
- You can call the `await` method of the `Condition` object to make th thread holding the lock wait and allow another thread to acquire the lock objects
- The thread with the condition is blocked until another thread executes the `signalAll` method on the same condition object

Synchronized Metods

- Earlier Java versions used synchronized methods
- Every Java object has one built-in lock and one built-in condition
- if a synchronized method is called, the lock is automatically created
- You can make any method synchronized by adding the `syncrhonized` keyword to the method header
  `public synchronized void deposit (double amount)`
- Synchronized methods automatically getthe lock/try/finally/unlock code implemented for it
- Conditions also work by using the `wait()` method

  ```
  while(balance < amount) wait();
  ```

  - Another method must call `notifyAll()` to end the wait
- Syncrhonized methods are easier, but more limited
  - Use these if they will work for your purpose