This week:
- multithreaded programming
- synchronized methods and blocks
- locks/monitors/mutexes
- `wait()`, `notify()`, `notifyAll()`

Web tutorial:
bank account example (1)

- one `BankAccount` object that allows withdrawals and deposits
- two `People` objects that are `Runnable` and that each have a reference to the same `BankAccount` object
public class BankAccount {
    private int balance;
    BankAccount(int b) { balance = b; }
    public int withdraw(int dollars) {
        if (balance >= dollars) {
            balance -= dollars;
            return dollars;
        }
        else throw new InsufficientBalanceException();
    }
    public void deposit(int dollars) {
        balance = balance + dollars;
    }
}
public static void main(String[] args) {
    BankAccount acc = new BankAccount(1000);
    // Boy, Girl both Runnable subclasses of People:
    Girl alice = new Girl(acc);
    Boy bob = new Boy(acc);
    Thread aliceThread = new Thread(alice);
    Thread bobThread = new Thread(bob);
    aliceThread.start();
    bobThread.start();
    // alice or bob can withdraw/deposit cash
    // at arbitrary times.
}
bank account example (3)

// in bob's run() method:
acc.deposit(100);

// in alice's run() method:
acc.withdraw(100);

// Assume bob gets interrupted here:
balance = balance + dollars;
   ^

// Here, rhs is 1100 dollars.
// Then alice takes over and withdraws 100 dollars
// (balance = 900). Then bob resumes and sets
// balance to 1100 dollars!
Moral: need to block withdrawals during deposits and vice versa. How?

- declare methods synchronized

```java
public synchronized int withdraw(int dollars) {
    if (balance >= dollars) {
        balance -= dollars;
        return dollars;
    }
    else throw new InsufficientBalanceException();
}
```

```java
public synchronized void deposit(int dollars) {
    balance = balance + dollars;
}
```
synchronized

- meaning of synchronized:
  - each object has a single lock (monitor, mutex) that can be held by only a single thread at a time
  - before entering a synchronized method
    - thread must acquire the lock
    - if it can't (already in use), must wait until lock released
  - lock is released when thread leaves synchronized method
rule of thumb

ANY method that
- manipulates the state of an object
- will be called from multiple threads
- should be synchronized

Q: why not declare ALL methods synchronized?
- slows things down
- acquiring/releasing lock takes time
synchronized blocks (1)

- Sometimes have methods like this:
  
  ```java
  public void foo() {
    // code that computes values independent
    //   of object state
    // code that manipulates state
  }
  ```

- could declare entire method synchronized
- wasteful (most of code doesn't manipulate state)
- how to declare part of method synchronized?
synchronized blocks (2)

```java
public void foo() {
    // code that computes values independent
    // of object state
    synchronized(this) { // grab lock of "this"
        // code that manipulates state
    }
}
```

- can also do synchronized blocks on other objects
cubbyhole example (1)

- a CubbyHole is an object that has a slot which contains at most one object
- one thread adds items into the CubbyHole
- one thread removes them from the CubbyHole
- CubbyHole empty $\to$ removing thread must wait until full before removing items
- CubbyHole full $\to$ adding thread must wait until empty before adding items
- want to add a series of items and withdraw in order
// Version 1:

public class CubbyHole {
    private Object slot;
    public CubbyHole() { slot = null; } // null == empty
    public void put(Object obj) { slot = obj; }
    public Object get() {
        Object obj = slot;
        slot = null;  // mark as empty
        return obj;
    }
}

cubbyhole example (3)

scenarios:

- one thread tries to `get()` when nothing there
  - returns null
- one thread tries to `put()` when slot occupied
  - object doesn't get transferred
- one thread tries to `put()` when another thread is `get()`-ting
  - object **might not** get transferred

all bad! Solution?
```java
// Version 2:
public class CubbyHole {
    private Object slot;
    public CubbyHole() { slot = null; } // null == empty
    public synchronized void put(Object obj) {
        slot = obj;
    }
    public synchronized Object get() {
        Object obj = slot;
        slot = null;  // mark as empty
        return obj;
    }
}
```
cubbyhole example (5)

- now can't \texttt{get()} and \texttt{put()} at same time, but what if...
  - one thread tries to \texttt{get()} when nothing there
  - one thread tries to \texttt{put()} when slot is full
- still broken!
- any ideas?
cubbyhole example (6)

// Version 3 (skip constructor, fields)
public synchronized void put(Object obj)
    throws InterruptedException {
    if (slot != null) { wait(); }
    slot = obj;
    notifyAll(); // signal that slot now filled
}

public synchronized Object get()
    throws InterruptedException {
    if (slot == null) { wait(); }
    Object obj = slot;
    slot = null; // mark as empty
    notifyAll(); // signal that slot now empty
    return obj;
}
cubbyhole example (7)

- ok for 1 producer thread, 1 consumer thread, but...
- imagine:
  - 2 producer threads, 2 consumer threads
  - one producer thread fills slot, calls `notifyAll()`
  - both consumer threads wake up
    - one grabs lock, removes item, returns (releasing lock)
    - other grabs lock, but no item to grab, so returns `null` (error!)
- moral: can't assume slot filled after returning from `wait()`
- solution?
public synchronized void put(Object obj) throws InterruptedException {
    while (slot != null) { wait(); }
    slot = obj;
    notifyAll();  // signal that slot now filled
}

public synchronized Object get() throws InterruptedException {
    while (slot == null) { wait(); }
    Object obj = slot;
    slot = null;  // mark as empty
    notifyAll();  // signal that slot now empty
    return obj;
}
assignment

- multithreaded version of web crawler
- need to use very similar pattern
  - actually somewhat easier – why?
- multithreaded programming _full_ of subtle traps like this
  - whole books, courses on how to avoid pitfalls
  - can't deal with it here
next week

- last assignment! woo hoo!
  - much easier

- serialization
  - saving precious objects to permanent storage and restoring them