Activity: What is the balance at the end?

Two threads both trying to `withdraw()` from the same account:

- Assume initial `balance` 150

```
class BankAccount {
  private int balance = 0;
  int  getBalance()      { return balance; }
  void setBalance(int x) { balance = x; }
  void withdraw(int amount) {
    int b = getBalance();
    if (amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
  }
  … // other operations like deposit, etc.
}
```

Thread 1

```
x.withdraw(100);
```

Thread 2

```
x.withdraw(75);
```


Activity: A bad interleaving

Interleaved `withdraw()` calls on the same account

- Assume initial `balance == 150`
- This ***should*** cause a `WithdrawTooLarge` exception

Time

Thread 1: `withdraw(100)`

```
int b = getBalance();




if (amount > b)
  throw new …;
setBalance(b – amount);
```

Thread 2: `withdraw(75)`

```
int b = getBalance();
if (amount > b)
  throw new …;
setBalance(b – amount);
```


*What we want: **Mutual exclusion***

The fix: Allow at most one thread to withdraw from account **A** at a time

- *Exclude* other simultaneous operations on **A** too (e.g., deposit)

Called **mutual exclusion**:

- One thread using a resource (here: a bank account) means another thread must wait
- We call the area of code that we want to have mutual exclusion (only one thread can be there at a time) a **critical section**.

Programmer (you!) must implement **critical sections**:

- "The compiler" has no idea what interleavings should or should not be allowed in your program
- But you need language primitives to do it!

Why is this Wrong?

Why can't we implement our own mutual-exclusion protocol?

- Say we tried to coordinate it ourselves using a boolean variable – "`busy`"
- It's technically possible under certain assumptions, but won't work in real languages anyway

```
class BankAccount {
  private int balance = 0;
  private boolean busy = false;
  void withdraw(int amount) {
    while (busy) { /* "spin-wait" */ }
    busy = true;
    int b = getBalance();
    if (amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
    busy = false;
  }
  // deposit would spin on same boolean
}
```

What we need

There are many ways out of this conundrum,
but we need help from the programming language…

One solution: **Mutual-Exclusion Locks** (aka Mutex, or just Lock)

- Still on a conceptual level at the moment, 'Lock' is not a Java class (though Java's approach is similar)

We will define Lock as an ADT with operations:

- `new`: make a new lock, initially *"not held"*
- `acquire`: blocks if this lock is already currently *"held"*
  - Once *"not held"*, makes lock *"held"* [all at once!]
  - Checking & setting happen together, and cannot be interrupted
  - Fixes problem we saw before!!
- `release`: makes this lock *"not held"*
  - If >= 1 threads are blocked on it, exactly 1 will acquire it

Note: 'Lock' is not an actual Java class

Almost-correct pseudocode

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();
  …
  void withdraw(int amount) {
    lk.acquire(); // may block
    int b = getBalance();
    if (amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
    lk.release();
  }
  // deposit would also acquire/release lk
}
```


Re-entrant lock

A re-entrant lock (a.k.a. recursive lock)

- "Remembers"
  - the thread (if any) that currently holds it
  - a *count*
- When the lock goes from *not-held* to *held*, the count is set to 0
- If (code running in) the current holder calls `acquire` :
  - it does not block
  - it increments the count
- On `release` :
  - if the count is > 0, the count is decremented
  - if the count is 0, the lock becomes *not-held*

`Synchronized`: A Java convenience

Java has built-in support for re-entrant locks

- You can use the `synchronized` statement as an alternative to declaring a `ReentrantLock`

```
synchronized (expression) {
  statements
}
```

1. Evaluates *expression* to an **object**
   - Every **object** (but not primitive types) "is a lock" in Java
2. Acquires the lock, blocking if necessary
   - "If you get past the `{`, you have the lock"
3. Releases the lock "at the matching `}`"
   - Even if control leaves due to `throw`, `return`, etc.
   - So *impossible* to forget to release the lock!