

# Concurrency and Mutual Exclusion

CSE 332 Summer 2020

**Instructor:** Richard Jiang

## Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

*Lecture Q&A: [pollev.com/332summer](https://pollev.com/332summer)*

*Lecture clarifications: [tinyurl.com/332-08-03A](https://tinyurl.com/332-08-03A)*

# Announcements

- ❖ Quiz #4 coming this week!
  - Quiz Group survey will be out today, fill it out by Wednesday 5pm
  - Contact your group ahead of time
  - If you fill out the survey, please participate
  
- ❖ Soft Deadline Reminders
  - Ex 10-11: Tonight
  - P3 Minimax: Tuesday

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-03A](https://tinyurl.com/332-08-03A)*

# Lecture Outline

- ❖ **Concurrency: Managing Correct Access to Shared Resources**
- ❖ Mutual Exclusion and Critical Sections
- ❖ Re-entrancy
- ❖ Locking in Java
- ❖ Race Conditions: Data Races vs. Bad Interleavings

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-08-03A](http://tinyurl.com/332-08-03A)*

# Concurrency: Canonical Example

- ❖ In a single-threaded world, this code is correct!

```
class BankAccount {
    private int balance = 0;

    protected int getBalance()      { return balance; }
    protected void setBalance(int x) { balance = x; }

    public void withdraw(int amount) {
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }

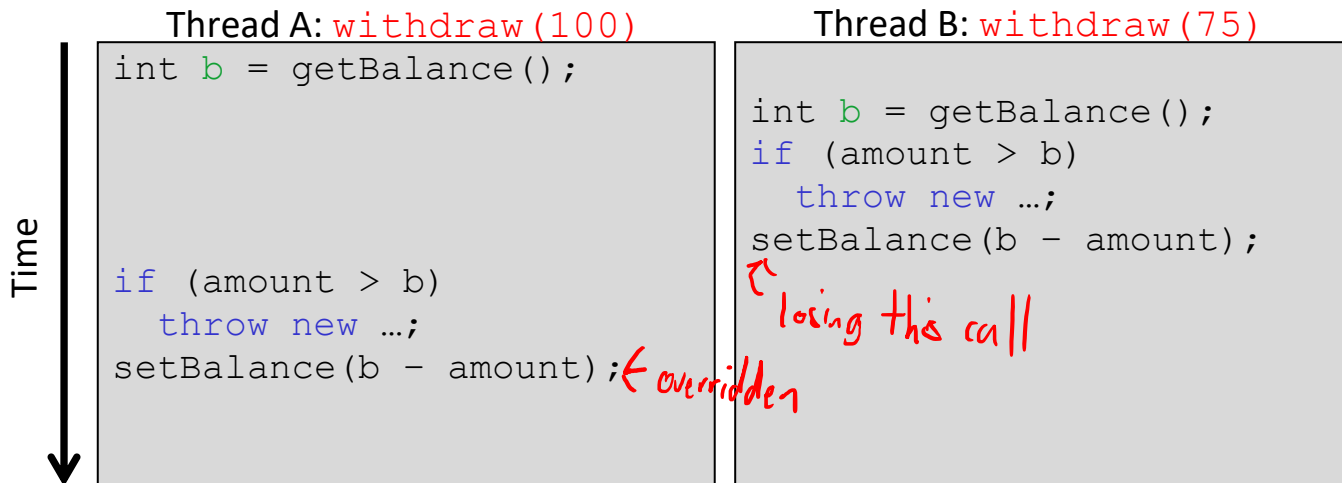
    // ... other operations like deposit(), etc.
}
```

# Interleaving

- ❖ Suppose:
  - Thread **T1** calls **`x.withdraw(100)`**
  - Thread **T2** calls **`y.withdraw(100)`**
- ❖ If second call starts before first finishes, we say they **interleave**
  - e.g. T1 runs for 50 ms, pauses somewhere, T2 picks up for 50ms
  - Can happen with one processor; if **pre-empted** due to time-slicing
- ❖ If **`x`** and **`y`** refer to different accounts, no problem
  - “You cook in your kitchen while I cook in mine”
  - But if **`x`** and **`y`** alias, possible trouble...

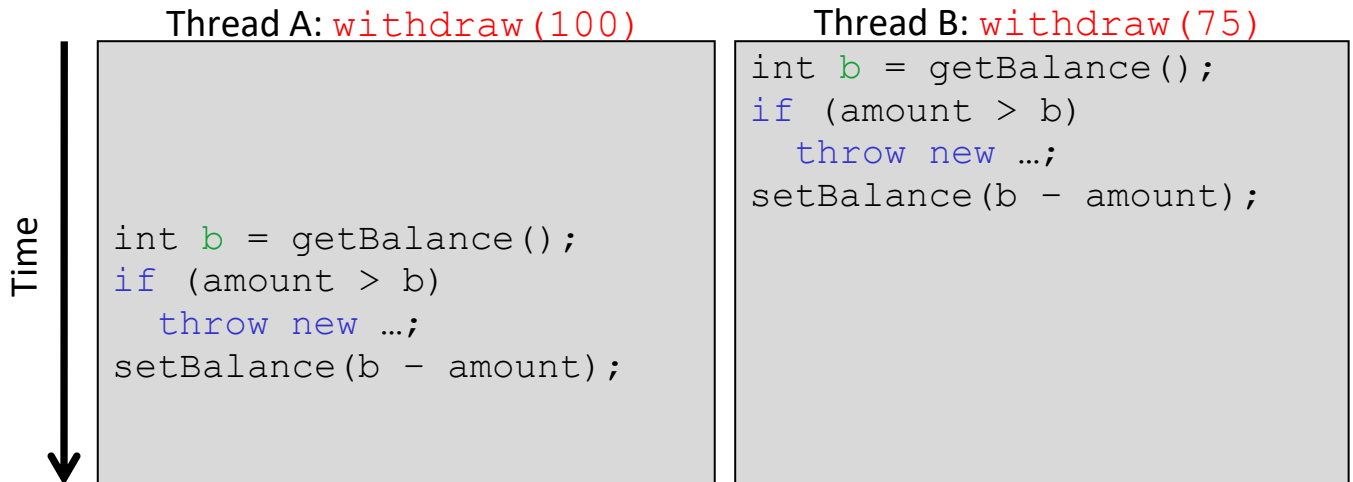
# A Bad Interleaving

- ❖ Interleaved `withdraw()` calls on the same account
  - Assume initial balance == 150
  - This *should* cause a `WithdrawTooLarge` exception (but doesn't)



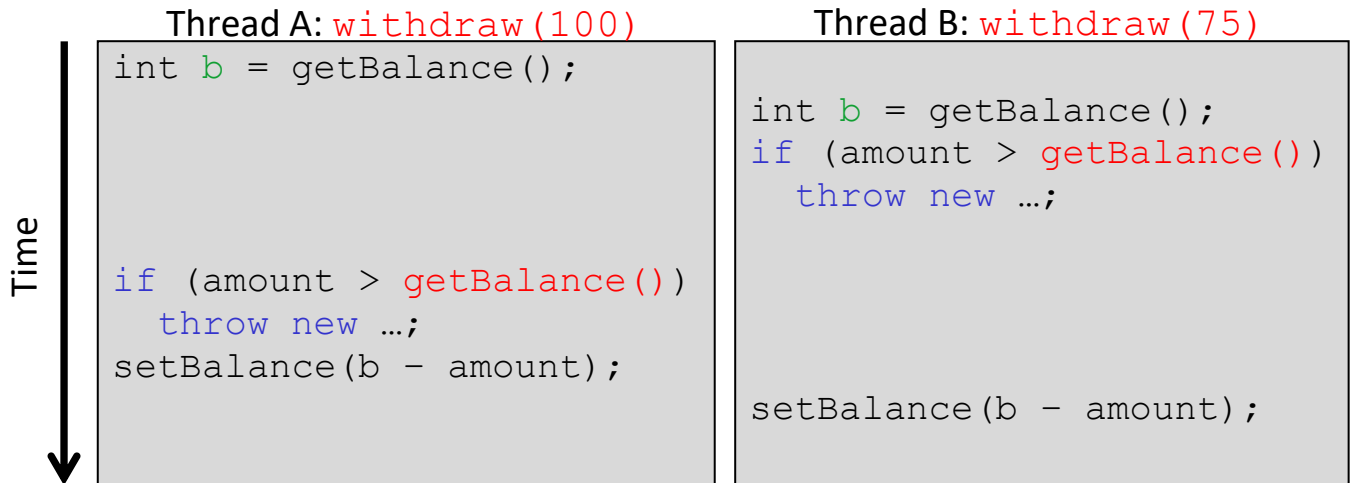
# A Good Interleaving is Also Possible

- ❖ Interleaved `withdraw()` calls on the same account
  - Assume initial balance == 150
  - This *does* cause a `WithdrawTooLarge` exception



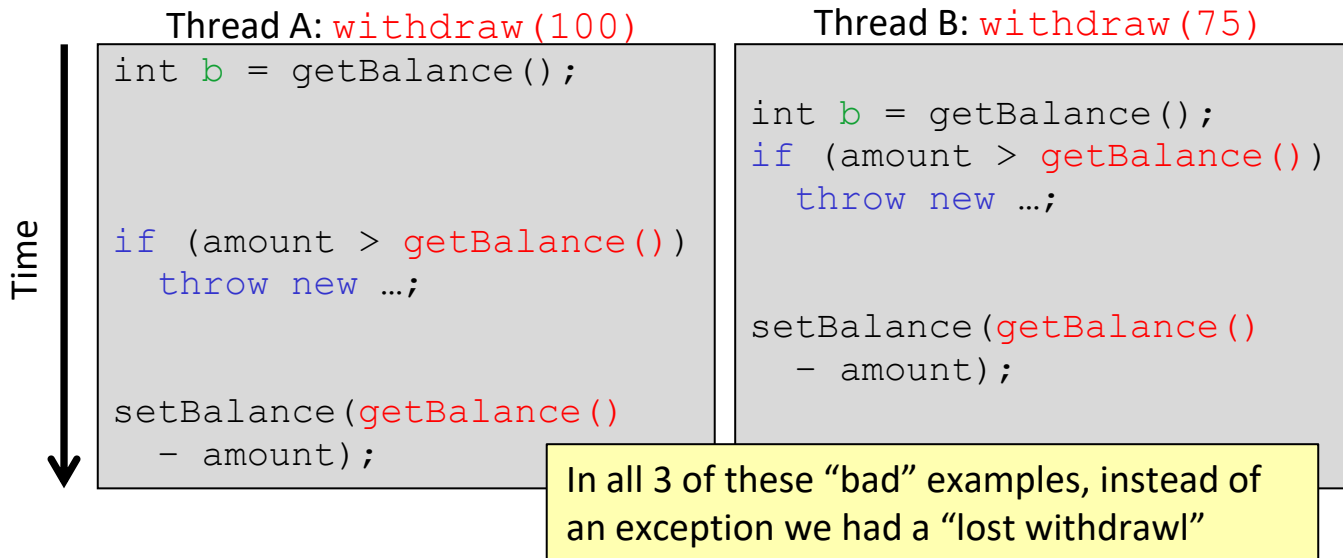
# A Bad Fix: Another Bad Interleaving

- ❖ Interleaved `withdraw()` calls on the same account
  - Assume initial balance == 150
  - This *should* cause a `WithdrawTooLarge` exception (but doesn't)



# ANOTHER Bad Fix: Another Bad Interleaving

- ❖ Interleaved `withdraw()` calls on the same account
  - Assume initial balance == 150
  - This *should* cause a `WithdrawTooLarge` exception (but doesn't)



# Incorrect “Fixes”

- ❖ It is tempting *and almost always wrong* to try fixing a bad interleaving by rearranging or repeating operations, such as:

```
public void withdraw(int amount) {  
    if (amount > getBalance())  
        throw new WithdrawTooLargeException();  
  
    // Maybe the balance was changed  
    setBalance(getBalance() - amount);  
}
```

- ❖ This fixes nothing!
  - Potentially narrows the problem by one statement
  - And that’s not even guaranteed!
    - The compiler could optimize it into the old version, because you didn’t indicate a need to synchronize

# Lecture Outline

- ❖ Concurrency: Managing Correct Access to Shared Resources
- ❖ **Mutual Exclusion and Critical Sections**
- ❖ Re-entrancy
- ❖ Locking in Java
- ❖ Race Conditions: Data Races vs. Bad Interleavings

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-03A](https://tinyurl.com/332-08-03A)*

# The Correct Fix: Mutual Exclusion

- ❖ Want at most one thread at a time to withdraw from account A
  - Exclude other simultaneous operations on A (e.g., deposit)
- ❖ More generally, we want **mutual exclusion**:
  - One thread using a resource means another thread must wait
- ❖ The area of code needing mutual exclusion is a **critical section**
  - *Only one thread*
- ❖ Programmer (you!) must identify and protect critical sections:
  - Compiler doesn't know which interleavings are allowed/disallowed
  - But you still need system-level primitives to do it!

# Why Do We Need System-level Primitives?

- ❖ Why can't we implement our own mutual-exclusion protocol?
  - Can we coordinate it ourselves using a boolean variable "**busy**"?
  - Possible under certain assumptions, but won't work in real languages

```
class BankAccount {
    private int balance = 0;
    private boolean busy = false;

    public 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
    }
}
```

*critical section*

# Because We Just Moved the Problem!

- Initially, **busy** = false

Thread A: `withdraw(100)`

```

while (busy) { } check
busy = true;      set
int b = getBalance();

if (amount > b)
    throw new ...;
setBalance(b - amount);
  
```

Thread B: `withdraw(75)`

```

while (busy) { }

busy = true;

int b = getBalance();
if (amount > b)
    throw new ...;
setBalance(b - amount);
  
```

Unhappy bank; we have a "lost withdrawal"

- Problem: time elapses between *checking and setting* **busy**
  - System can interrupt a thread then, letting another thread "sneak in"

# What We Actually Need: Lock ADT

- ❖ All ways out of this conundrum require system-level support
- ❖ One solution: **Mutual-Exclusion Locks** (aka **Mutex**, or just **Lock**)
  - For now, still discussing concepts; `Lock` is not a Java class
- ❖ We will define **Lock** as an **ADT** with operations:
  - **new**: make a new lock, initially “not held”
  - **acquire**: blocks current thread if this lock is “held”
    - Once “not held”, makes lock “held”
    - Checking & setting the “held” boolean is a single uninterruptible operation
    - Fixes problem we saw before!!
  - **release**: makes this lock “not held”
    - If  $\geq 1$  threads are blocked on it, another thread – but only one! – can now acquire

*atomic*

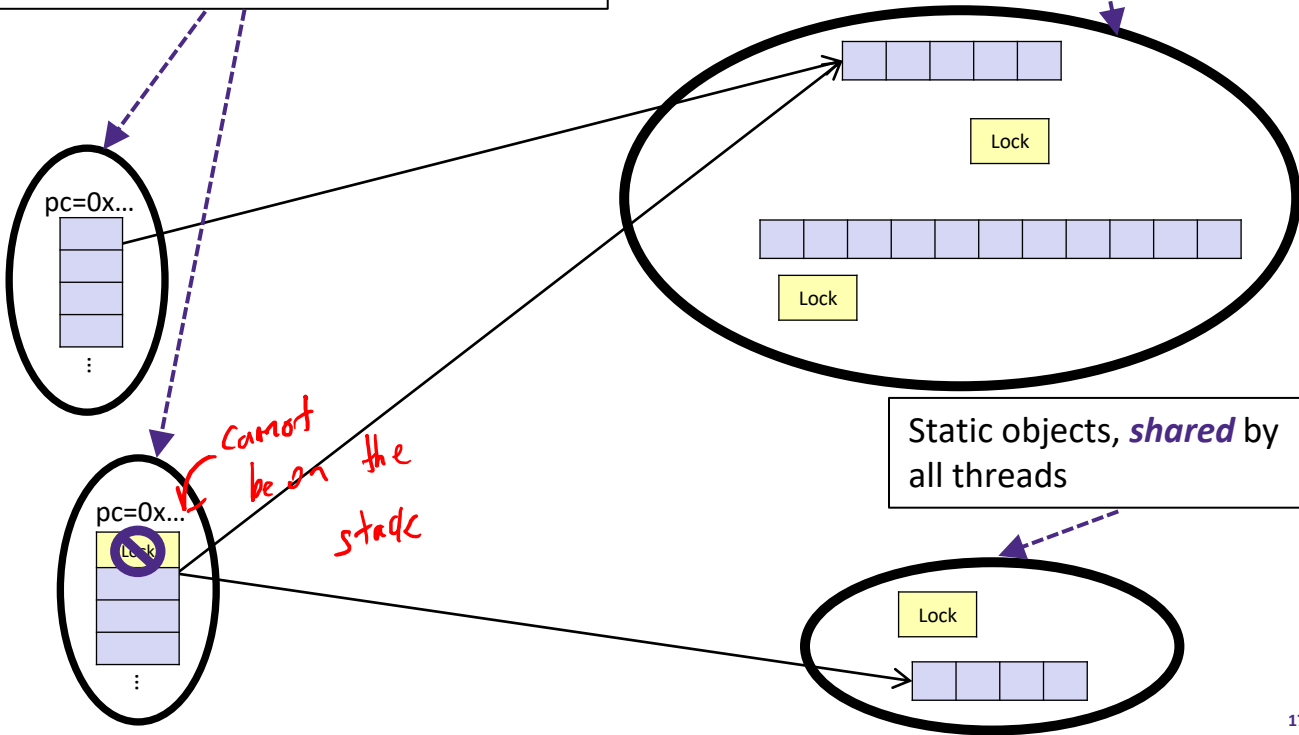
# Why a System-level Lock Works

- ❖ Lock must ensure that, given simultaneous acquires/releases, “the correct thing” will happen
  - E.g.: if we have two acquires: one will “win” and one will block
- ❖ How can this be implemented?
  - The key is that the “check if held; if not, make held” operation must happen “all-at-once”. It cannot be interrupted!
  - Thus, requires and uses hardware and O/S support
    - See computer-architecture or operating-systems course
  - In CSE 332, we’ll assume a lock is a primitive and just use it

# Locks Must Be Accessible By Multiple Threads!

Multiple threads, each with its own *independent* call stack and program counter

Heap for allocated objects, *shared* by all threads



# Almost-Correct Pseudocode

```
class BankAccount {
    private int balance = 0;
    private Lock lk = new Lock();

    public 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
}
```

Note: 'Lock' is not an actual Java class

# Activity: Questions About the Previous Slide

1. Where is the critical section?
2. How many locks do we need?
  - a) One lock per BankAccount object?
  - b) Two locks per BankAccount object? *withdraw and deposit use balance*
    - i.e., one for withdraw() and one for deposit()
  - c) One lock for the entire Bank *correct, but low concurrency*
    - Bank contains multiple BankAccount instances
3. There is a bug in withdraw(), can you find it?  
✓
4. Do we need locks for:
  - a) getBalance?
  - b) setBalance? *→ reentry*

# Answers: Some Common Locking Mistakes

- ❖ A lock is very primitive; up to you to use correctly
- ❖ **Incorrect**: different locks for **withdraw** and **deposit**
  - Mutual exclusion works only when sharing same lock
  - **balance** field is the shared resource being protected
- ❖ **Poor performance**: same lock for entire Bank
  - No simultaneous operations on *different* accounts (low **concurrency**)
- ❖ **Bug**: forgot to release a lock when exiting early
  - Can block other threads forever if there's an exception

```
if (amount > b) {  
    lk.release(); // hard to remember!  
    throw new WithdrawTooLargeException();  
}
```

Remembering to release() before every exit is challenging!

# Summary of Mutual Exclusion


- ❖ Threads are useful beyond just fork-join-style parallelism
  - But general use-cases require **concurrency** to ensure correctness when dealing with overlapped sharing
- ❖ Overlapped sharing introduces **non-determinism** because the system controls the scheduling of threads
  - Therefore, the system must also provide **locks** to ensure **mutual exclusion** in **critical sections** of code
  - Mutual exclusion is the technique we employ to prevent **bad interleavings**

# Lecture Outline

- ❖ Concurrency: Managing Correct Access to Shared Resources
- ❖ Mutual Exclusion and Critical Sections
- ❖ **Re-entrancy**
- ❖ Locking in Java
- ❖ Race Conditions: Data Races vs. Bad Interleavings

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-03A](https://tinyurl.com/332-08-03A)*

# Adding Operations to Our BankAccount

- ❖ If **withdraw** and **deposit** use the same lock, then simultaneous calls to them are properly synchronized 
- ❖ But what about **getBalance** and **setBalance**?
  - Assume now that they are **public** (which may be reasonable)
  - If they **do not acquire the same lock**, then a race between **setBalance** and **withdraw** could produce a wrong result
  - If they **do acquire the same lock**, then **withdraw** would block forever because it tries to acquire a lock it already has!

# One (Not Very Good) Possibility

- ❖ Have two versions of setBalance!
  - withdraw() calls a non-locking version of setBalance() (since it already has the lock)
  - Outside world calls the locking version of setBalance()
- ❖ Could work if adhered to, but not good style
  - Also inconvenient
- ❖ Alternately, we can modify the meaning of the Lock ADT to support **re-entrant locks**
  - Java does this
  - Then just always use the locking version of setBalance()

```
private int setBalanceNoLock(  
    int x) {  
    balance = x;  
}  
  
public int setBalance(int x) {  
    lk.acquire();  
    critical [ setBalanceNoLock(x)  
    lk.release();  
}  
  
public void withdraw(int amount) {  
    lk.acquire();  
    critical [  
    ...  
    setBalanceNoLock(b - amount);  
    lk.release();  
}
```

# Re-entrancy

$T_1 \rightarrow 2$

- ❖ A **re-entrant lock** (a.k.a. **recursive lock**)
  - Once acquired, the lock is held *by the executor*,
  - Subsequent acquire calls *in that executor* won't block
  
- ❖ Example:
  - `withdraw()` can acquire the lock
  - Then, `withdraw()` can call `setBalance()`, which also acquires the lock
  - Because they're *in the same executor* and it's a *re-entrant* lock, the inner acquire won't block! *thread*

# Re-entrant Lock Implementation

- ❖ Contains the following state:
  - the thread (if any) that currently holds it and a count
- ❖ When the lock goes from not-held to held:
  - remembers the thread and sets count = 0
- ❖ If the *current holder* calls `acquire()` again:
  - it does not block and `count++`
- ❖ If the *current holder* calls `release()`:
  - if `count > 0` and `count--`
  - if `count == 0`, the lock “forgets” the thread

# Lecture Outline

- ❖ Concurrency: Managing Correct Access to Shared Resources
- ❖ Mutual Exclusion and Critical Sections
- ❖ Re-entrancy
- ❖ **Locking in Java**
- ❖ Race Conditions: Data Races vs. Bad Interleavings

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-08-03A](http://tinyurl.com/332-08-03A)*

# Java's Re-entrant Lock

- ❖ Java doesn't have the "plain" lock we discussed earlier; it only has re-entrant locks
- ❖ `java.util.concurrent.locks.ReentrantLock`
  - Has methods `lock()` and `unlock()`

# Locking Best Practices in Java

- ❖ Remember our bug in withdraw()?

*Cover all exits*

- ❖ Need to guarantee that locks are always released

- Recommend something like this:

```
myLock.lock(); acquire
```

```
try { /* method body */ }
```

```
finally { myLock.unlock(); } will always execute
```

*release*

- The code in `finally` will always execute afterwards
  - Regardless of exceptions, returns, or “normal” completion

# synchronized: A Java Convenience

- ❖ Or use `synchronized` statement instead of explicitly instantiating a `ReentrantLock` + `try/catch/finally` blocks

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

*critical* → *release* (pointing to the closing brace)  
*acquire* (pointing to the opening brace)

- ❖ `synchronized` statement:
  - Evaluates *expression* to an object
    - Every object (but not primitive types) can be a lock in Java
  - Acquires the lock, blocking if necessary
    - “If you get past the {, you have the lock”
  - Releases the lock “at the matching }”, even if throw, return, etc.
    - So it’s impossible to forget to release the lock

## Version #1: Correct, But Can Be Improved

```
class BankAccount {
    private int balance = 0;
    private Object lk = new Object();

    protected int getBalance()
        { synchronized (lk) { return balance; } }
    protected void setBalance(int x)
        { synchronized (lk) { balance = x; } }

    public void withdraw(int amount) {
        synchronized (lk) {
            int b = getBalance();
            if (amount > b)
                throw ...
            setBalance(b - amount);
        }
    }

    // deposit() would also use synchronized(lk)
}
```

# Improving Version #1

- ❖ As written, the lock is **private**
  - Seems like a good idea ... ?
  - But prevents other classes from synchronizing with BankAccount operations
- ❖ More idiomatic is to synchronize on **this**
  - Also more convenient: no need to have an extra object!

## Version #2: Still Improvable

```
class BankAccount {
    private int balance = 0;

    protected int getBalance()
    { synchronized (this){ return balance; } }
    protected void setBalance(int x)
    { synchronized (this){ balance = x; } }

    public void withdraw(int amount) {
        synchronized (this) {
            int b = getBalance();
            if(amount > b)
                throw ...
            setBalance(b - amount);
        }
    }

    // deposit() would also use synchronized(this)
}
```

*Synchronize (BankAccount)*

## Improving Version #2: Syntactic Sugar

- ❖ There is a shorter way to say the same thing as version #2
- ❖ Putting **synchronized** before a method declaration means the entire method body is surrounded by



```
synchronized(this) {...}
```

- ❖ Version #3 is *identical to version #2*, but more concise, more standard, and therefore better style

## Version #3: Final Version

```
class BankAccount {  
    private int balance = 0;  
  
    synchronized protected int getBalance()  
    { return balance; }  
    synchronized protected void setBalance(int x)  
    { balance = x; }  
  
    synchronized public void withdraw(int amount) {  
        int b = getBalance();  
        if (amount > b)  
            throw ...  
        setBalance(b - amount);  
    }  
  
    // deposit() would also use synchronized  
}
```

*still synchronized (this)*

# A Few Final Thoughts

- ❖ Our `synchronized`-less `BankAccount` pseudocode needs `java.util.concurrent.locks.ReentrantLock` and `try { ... } finally { ... }` blocks
  - Or just used `synchronized` 😊
- ❖ Don't have time to cover all highly-relevant lock variants: *readers/writer locks, condition variables, etc.*
  - A little more info on readers/writer locks on next slide..
- ❖ Java provides many other features and details. See also:
  - Chapter 14 of *CoreJava, Volume 1* by Horstmann/Cornell
  - *Java Concurrency in Practice* by Goetz et al

# A Few Final Thoughts

- ❖ **Readers/Writer Locks** allow more **concurrency**
  - Simultaneous reads are often okay so we can let many readers have the lock
  - We can still only have one writer
  - Threads acquire either a reader or writer lock
  - Only one lock can be active at a time
    - Many readers can hold the reader lock
    - One writer can hold the writer lock

acquire Reader()  
acquire Writer()

# Lecture Outline

- ❖ Concurrency: Managing Correct Access to Shared Resources
- ❖ Mutual Exclusion and Critical Sections
- ❖ Re-entrancy
- ❖ Locking in Java
- ❖ **Race Conditions: Data Races vs. Bad Interleavings**

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-03A](https://tinyurl.com/332-08-03A)*

# Race Conditions

- ❖ A **race condition** occurs when the computation result depends on scheduling (ie, how threads are interleaved)
  - i.e.: if T1 and T2 are scheduled in a certain way, things go “wrong”
  - Only exist due to concurrency: no interleaving with only 1 thread!
- ❖ We, as programmers, cannot control scheduling of threads
  - Thus we must write programs that work independent of scheduling

balance = 150    withdraw(100)    withdraw(75)

1 outcome → balance = 50    w(100) ✓    w(75) ✗

2nd outcome → balance = 75    w(100) ✗    w(75) ✓

# Data Races vs. Bad Interleavings

- ❖ We will make a big distinction between:

*data races*      and      *bad interleavings*

- ❖ Both are types of *race conditions*
  - Confusion often results from not distinguishing these, or using the term “race condition” to refer to only one of these two

## Very Briefly: Data Races

- ❖ A **data race** is a type of **race condition** that can happen when:
  - Different threads **potentially** write a variable *at the same time*
  - One thread **potentially** writes a variable *at the same time* another thread reads it

w/w    r/r    w/r

- ❖ Two threads *reading* the same variable at the same time is not a data race and doesn't create an error
  - The key is that one of the threads must be *writing* to the variable
- ❖ The 'potentially' is important!
  - Code has a data race independent of any particular actual execution

# Bad Interleavings

- ❖ Easy to see why **data races** are bad
- ❖ However, we can still have a **race condition** (and bad behavior) even without **data races**, thanks to **bad interleavings**
  - Different threads' reads and writes are “interleaved” without simultaneity



- ❖ Warning sign: intermediate/temporary state visible to a concurrently executing thread
  - E.g.: partial insert in a linked list: ‘front’ field updated with new node, but ‘count’ not yet updated