UNIVERSITY OF WASHINGTON 1861

CSE332: Data Abstractions

Lecture 19: Mutual Exclusion and Locking

James Fogarty
Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman

Banking Example

This code is correct in a single-threaded world

```
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 InsufficientFundsException();
    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 the calls interleave

- Could happen even with one processor,
  as a thread can be pre-empted for time-slicing
  (e.g., T1 runs for 50 ms, T2 runs for 50ms, T1 resumes)

If **`x`** and **`y`** refer to different accounts, no problem

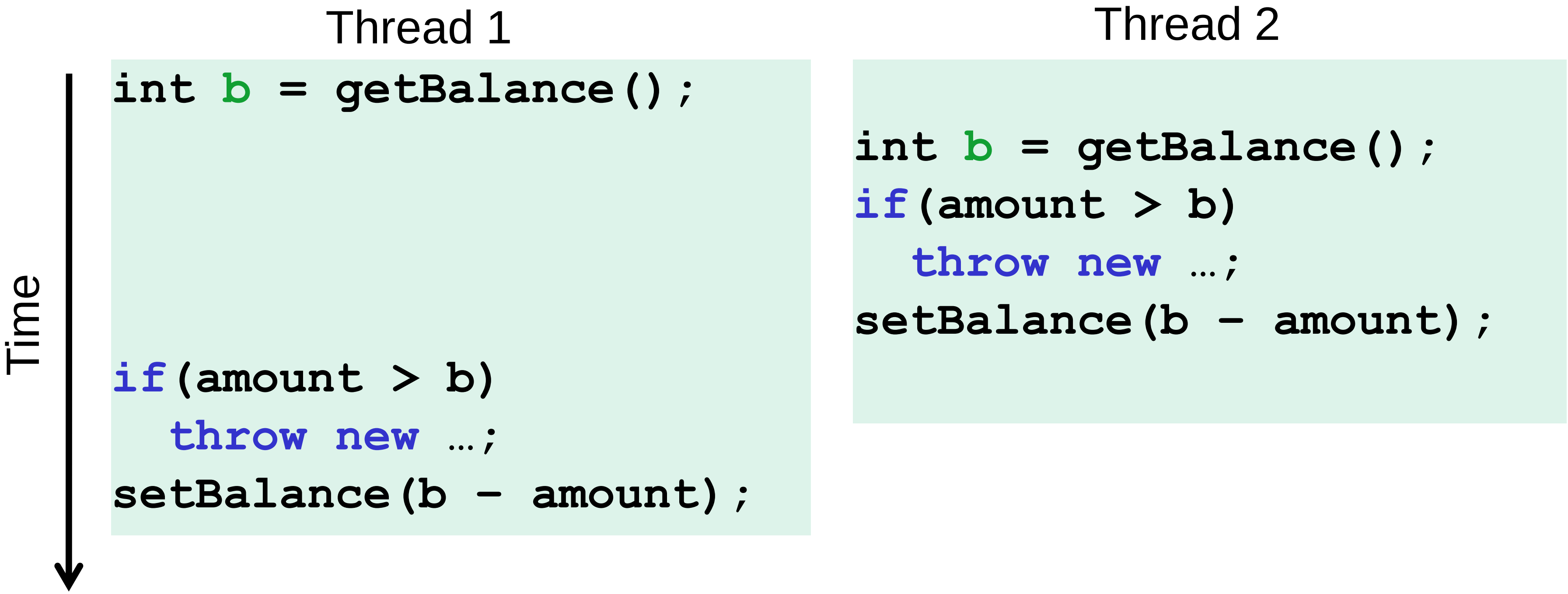
- “You cook in your kitchen while I cook in mine”

But if **`x`** and **`y`** alias, possible trouble…

Bad Interleaving

Interleaved **`withdraw(100)`** calls on the same account

- Assume initial **`balance == 150`**



Thread 1

```
int b = getBalance();




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

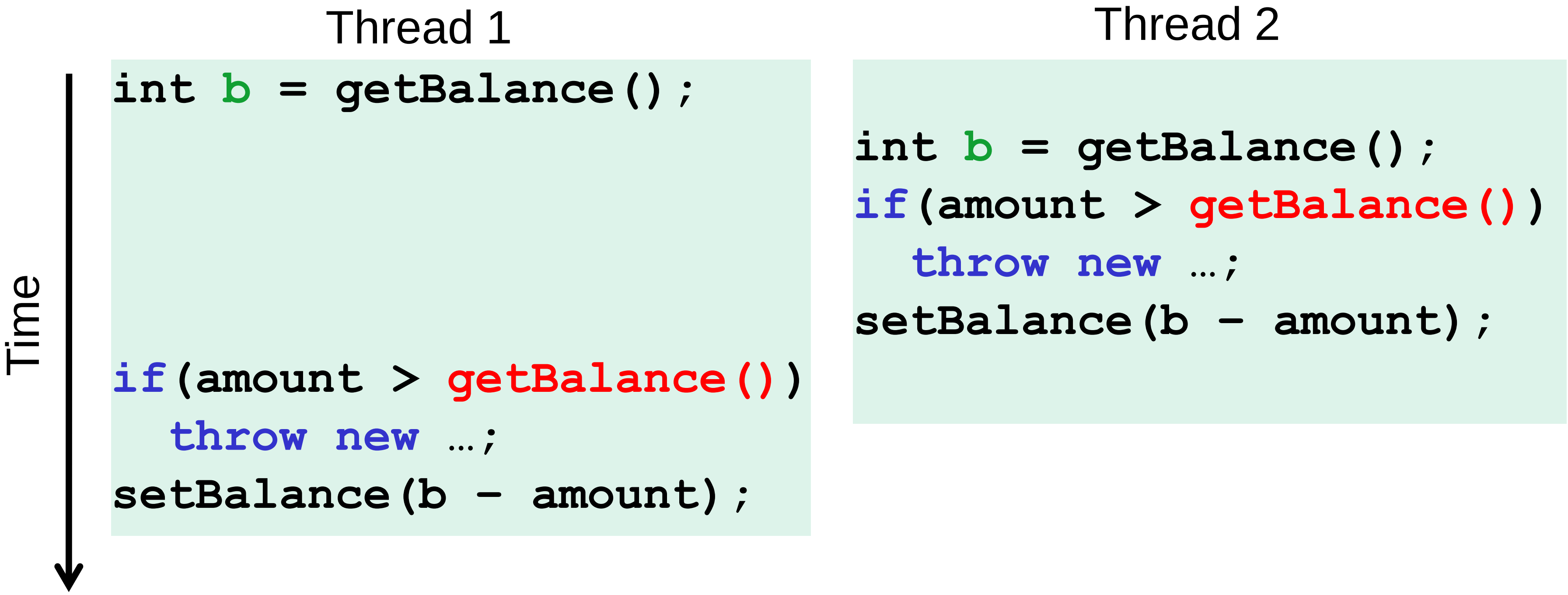
Thread 2

```

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

Incorrect Attempt to “Fix”

Interleaved **`withdraw(100)`** calls on the same account

- Assume initial **`balance == 150`**



Thread 1

```
int b = getBalance();




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

Thread 2

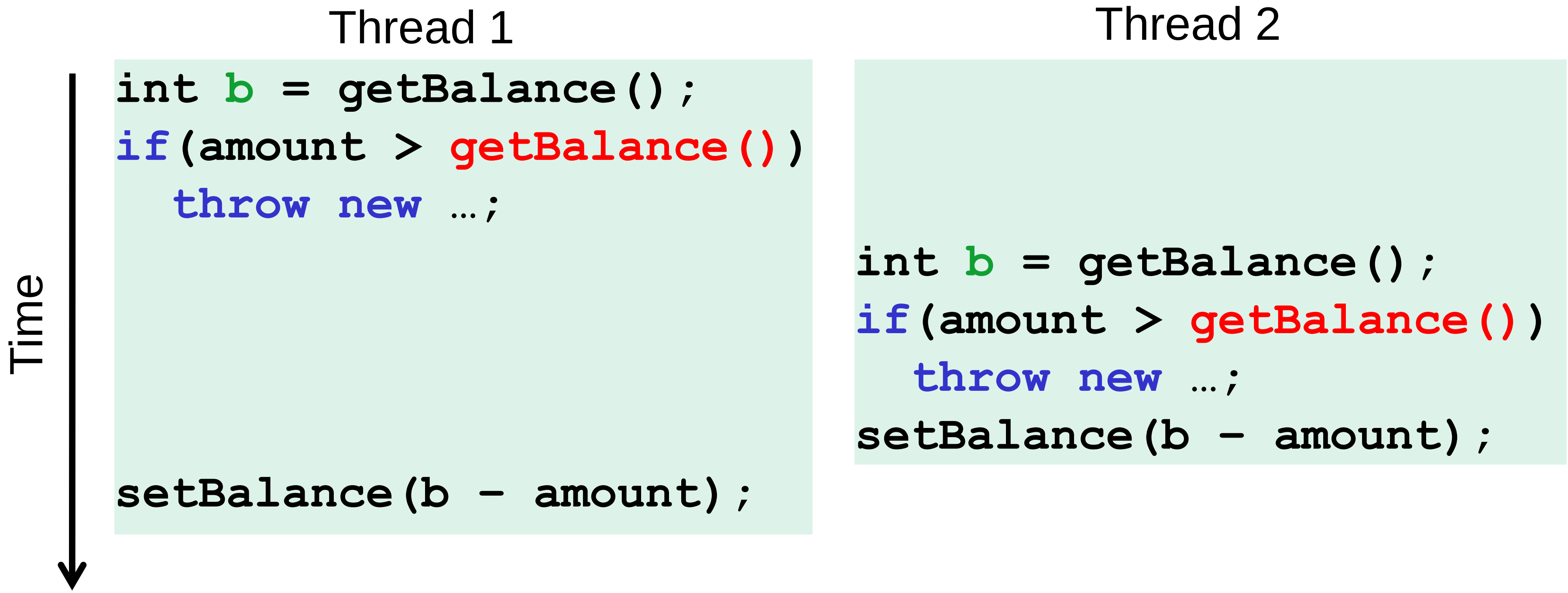
```

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

This interleaving would throw an exception

Incorrect Attempt to “Fix”

Interleaved `withdraw(100)` calls on the same account

  – Assume initial `balance == 150`



Thread 1

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




setBalance(b – amount);
```

Thread 2

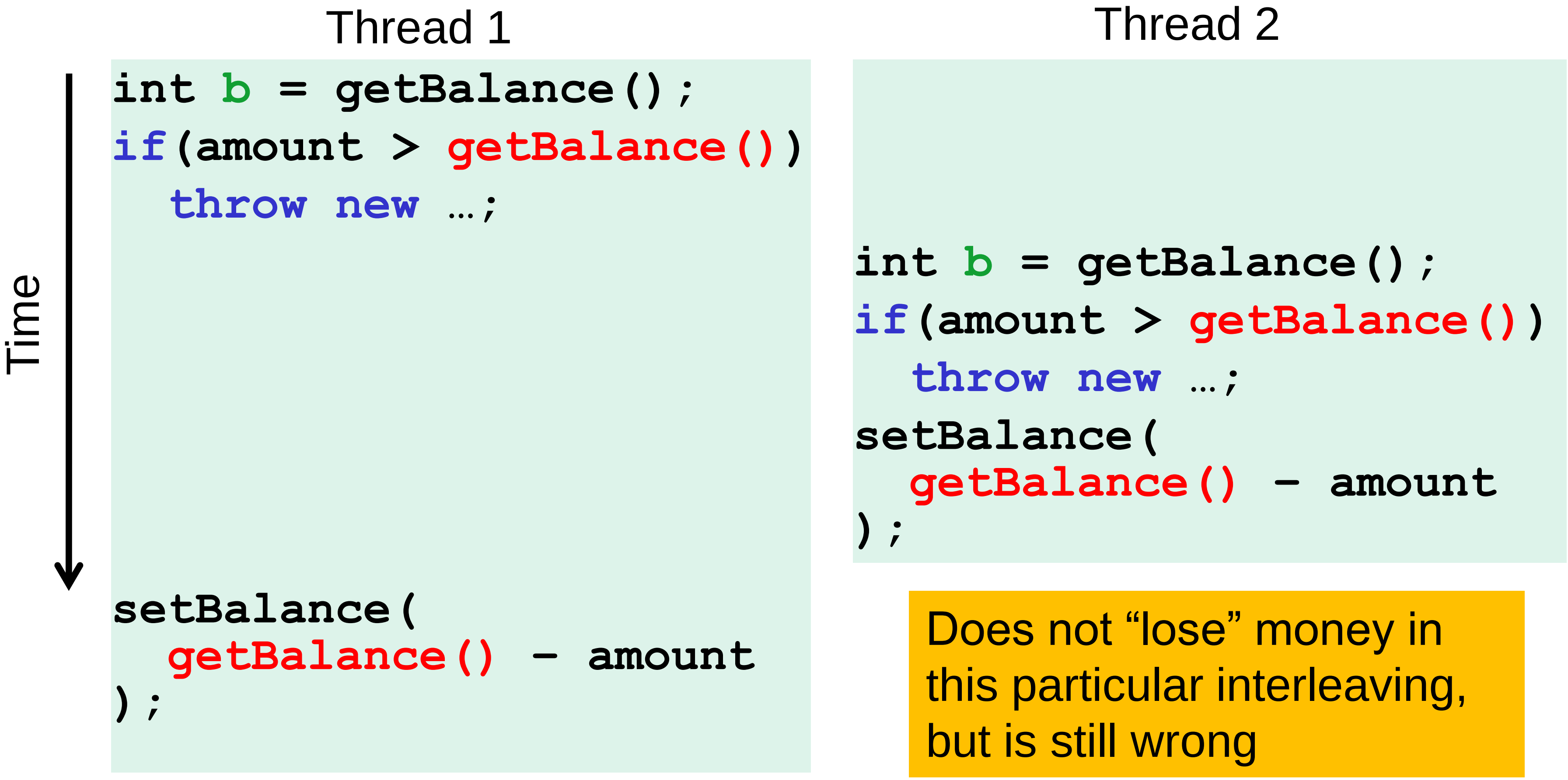
```



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

But this interleaving loses the withdrawal

Incorrect Attempt to “Fix”

Interleaved **`withdraw(100)`** calls on the same account

- Assume initial **`balance == 150`**



Thread 1

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

Thread 2

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

Thread 1

```
setBalance(
  getBalance() - amount
);
```

Does not “lose” money in this particular interleaving, but is still wrong

Incorrect Attempt to “Fix”

It can be tempting, but is generally wrong, to attempt
to “fix” a bad interleaving by rearranging or repeating operations

```
void withdraw(int amount) {
  if(amount > getBalance())
    throw new InsufficientFundsException();
  // maybe balance changed
  setBalance(getBalance() – amount);
}
```

Only narrows the problem by one statement

- Imagine a withdrawal is interleaved after computing the value of the parameter `getBalance() – amount` but before invocation of the function `setBalance`

Your compiler might even remove the second call to `getBalance()`, because you have not told it you need to synchronize

Mutual Exclusion

The sane fix is to allow only one thread withdrawing from **A** at a time

- Also exclude other simultaneous operations on **A** that could potentially result in bad interleavings (e.g., deposit)

Mutual exclusion: One thread doing something with a resource means that any other thread must wait until the resource is available

- Define critical sections; areas of code that are mutually exclusive

Programmer must implement critical sections

- “The compiler” has no idea what interleavings should or should not be allowed in your program
- But you will need language primitives to do this

Incorrect Attempt to “Do it Ourselves”

```
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 InsufficientFundsException();
    setBalance(b – amount);
    busy = false;
  }
  // deposit would spin on same boolean
}
```

This Just Moves the Problem

Time

Thread 1

```
while(busy) { }

busy = true;

int b = getBalance();




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

Thread 2

```

while(busy) { }

busy = true;

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

Need Help from the Language

- There are many ways out of this conundrum

- One basic solution: Locks
  - Still on a conceptual, ‘Lock’ is not a Java class

- We will define Lock as an ADT with operations:
  - **`new`**: make a new lock
  - **`acquire`**: If lock is *“not held”*, makes it *“held”*
    - Blocks if this lock is already currently *“held”*
    - Checking & Setting happen atomically, cannot be interrupted
      - Details of that require hardware and system support
  - **`release`**: makes this lock *“not held”*
    - if >= 1 threads are blocked on it, exactly 1 will acquire it

Still Incorrect 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 InsufficientFundsException();
    setBalance(b – amount);
    lk.release();
  }
  // deposit would also acquire/release lk
}
```

Some Mistakes

- A lock is a very primitive mechanism
  - Still must be used correctly to implement critical sections
- **Incorrect**: Forget to release a lock, thus blocks other threads forever
  - Previous slide is wrong because of the exception possibility

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

- **Incorrect** : Use different locks for `withdraw` and `deposit`
  - Mutual exclusion works only when using same lock
  - Balance is the shared resource that is being protected
- **Poor performance**: Use same lock for every bank account
  - No simultaneous withdrawals from *different* accounts

Other Operations

- If `withdraw` and `deposit` use the same lock, then simultaneous calls to these methods are properly synchronized
- But what about `getBalance` and `setBalance`
  - Assume 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

```
…
     lk.acquire();
     int b = getBalance();
…
```

One Bad Option

```
int setBalanceUnsafe(int x) {
  balance = x;
}
int setBalanceSafe(int x) {
  lk.acquire();
  balance = x;
  lk.release();
}
void withdraw(int amount) {
  lk.acquire();
  …
  setBalanceUnsafe(b – amount);
  lk.release();
}
```

Two versions of setBalance

- Safe and unsafe versions
- Use one or the other, depending on whether you already have the lock

Technically could work

- Hard to always remember
- And definitely poor style

Better to modify meaning of the Lock ADT to support ***re-entrant locks***

Re-Entrant Locking

A re-entrant lock is also known as a recursive lock

  - “Remembers” the thread that currently holds it
  - Stores a *count* of “how many” times it is held

- When lock goes from ***not-held*** to ***held***, the count is set to 0
- If 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***

`withdraw` can acquire the lock, and then call **`setBalance`**

Java’s Re-Entrant Lock

`java.util.concurrent.locks.ReentrantLock`

- Has methods **`lock()`** and **`unlock()`**

Important to guarantee that lock is always released

```
myLock.lock();
try {
      // method body
} finally {
      myLock.unlock();
}
```

Regardless what happens in ‘try’, finally code will execute

A Java Convenience: `Synchronized`

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** “is a lock” in Java (but not primitive types)
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`, or whatever
   - So it is impossible to forget to release the lock

Java Version #1: Correct but not “Java Style”

```
class BankAccount {
  private int balance = 0;
  private Object lk = new Object();
  int getBalance()
    { synchronized (lk) { return balance; } }
  void setBalance(int x)
    { synchronized (lk) { balance = x; } }
  void withdraw(int amount) {
    synchronized (lk) {
      int b = getBalance();
      if(amount > b)
        throw …
      setBalance(b – amount);
    }
  }
  // deposit would also use synchronized(lk)
}
```

Improving the Java

- As written, the lock is **private**
  - Might seem like a good idea
  - But also prevents code in other classes from writing operations that synchronize with the account operations

- Example motivations with our bank record?

- It is more common to synchronize on **`this`**
  - It is also convenient; no need to declare an extra object

Java Version #2: Still Missing Sugar

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

Syntactic Sugar

Java provides a concise and standard way to say the same thing:

Applying the `synchronized` keyword to a method declaration means the entire method body is surrounded by

```
synchronized(this){
    …
}
```

Next version means exactly the same thing,
but is more concise and more the “style of Java”

Java Version #3: Final Version

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

Races

A race condition occurs when the computation result depends on scheduling (how threads are interleaved on one or more processors)

- If T1 and T2 are scheduled in a particular way, then things go wrong
- As programmers, we cannot control scheduling of threads;
  we need to write programs that are correct independent of scheduling

Race conditions are bugs that exist only due to concurrency

- No interleaved scheduling with 1 thread

Typically, the problem is some *intermediate state* that
“messes up” a concurrent thread that “sees” that state

We will distinguish between data races and bad interleavings,
both of which are types of race condition bugs

Data Races

- A data race is a type of ***race condition*** that can happen in 2 ways:
  - Two threads can ***potentially*** write a variable at the same time
  - One thread can ***potentially*** write a variable while another reads it

- Simultaneous reads are fine; not a data race, and no bad results

- 'Potentially' is important; we say the code itself has a data race
  - This is independent of any particular actual execution

- Data races are bad, but are not the only form of race condition
  - We can have a race, and bad behavior, without any data race

Stack Example

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  synchronized boolean isEmpty() {
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    if(isEmpty())
      throw new StackEmptyException();
    return array[index--];
  }
}
```

A Race Condition: But Not a Data Race

```
class Stack<E> {
  …
  synchronized boolean isEmpty() { … }
  synchronized void push(E val) { … }
  synchronized E pop(E val) {
    if(isEmpty())
      throw new StackEmptyException();
    …
  }
}

E peek() {
  E ans = pop();
  push(ans);
  return ans;
}
```

- In a sequential world, this code is of questionable *style*, but *correct*
- The “algorithm” is the only way to write a **`peek`** helper method if all you have is this interface

Examining **`peek`** *in a Concurrent Context*

- **`peek`** has no *overall* effect on the shared data
  - It is a “reader” not a “writer”
  - State should be the same after it executes as before

- This implementation creates an inconsistent *intermediate state*
  - Calls to **`push`** and **`pop`** are synchronized,
    so there are no ***data races*** on the underlying array
  - But there is still a ***race condition***

- This intermediate state should not be exposed
  - Leads to several *bad interleavings*

```
E peek() {
      E ans = pop();
      push(ans);
      return ans;
}
```

Example 1: peek and isEmpty

- **Property we want**:
  If there has been a **`push`** (and no **`pop)`**,
  then **`isEmpty`** should return **`false`**

- With **`peek`** as written, property can be violated – how?

Thread 1 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
boolean b = isEmpty()
```

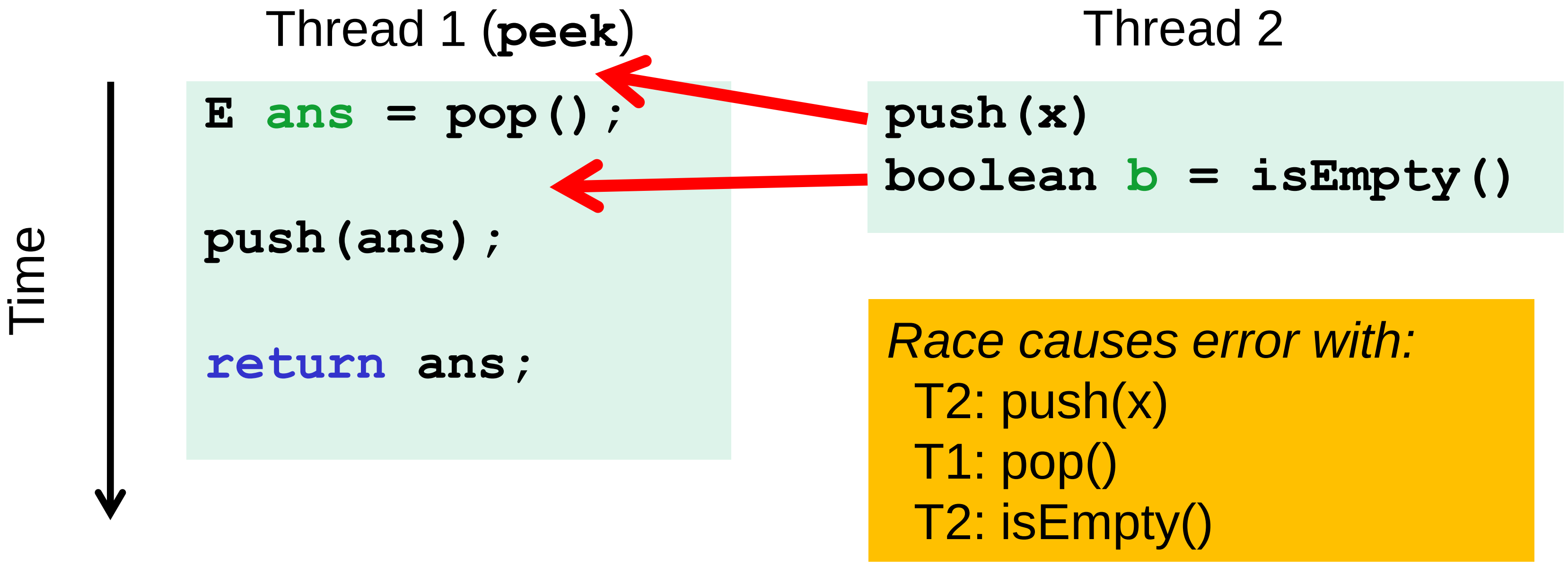
Time

Example 1: peek and isEmpty

- **Property we want**:

  If there has been a **`push`** (and no **`pop)`**,
  then **`isEmpty`** should return **`false`**

- With **`peek`** as written, property can be violated – how?



Thread 1 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

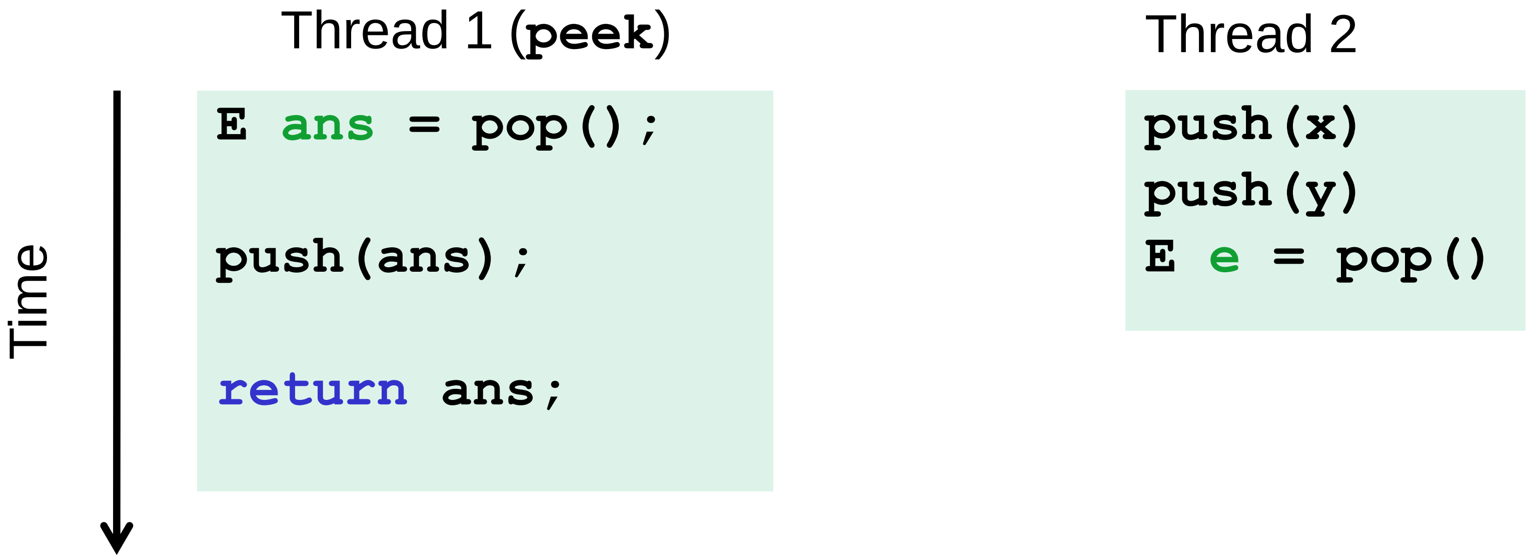
```
push(x)
boolean b = isEmpty()
```

Race causes error with:
T2: push(x)
T1: pop()
T2: isEmpty()

Example 2: peek and push

- **Property we want:**
  Values are returned from **`pop`** in LIFO order

- With **`peek`** as written, property can be violated – how?



Thread 1 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

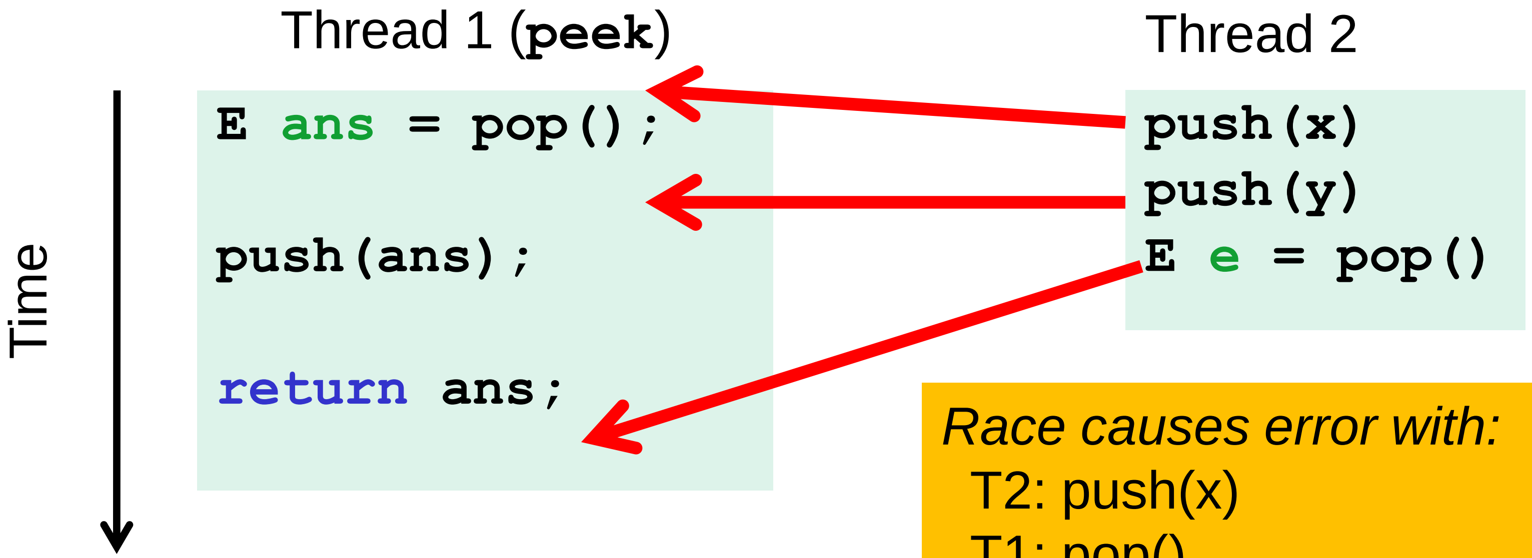
Thread 2

```
push(x)
push(y)
E e = pop()
```

Example 2: peek and push

- **Property we want:**
  Values are returned from `pop` in LIFO order

- With `peek` as written, property can be violated – how?



Thread 1 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
push(y)
E e = pop()
```

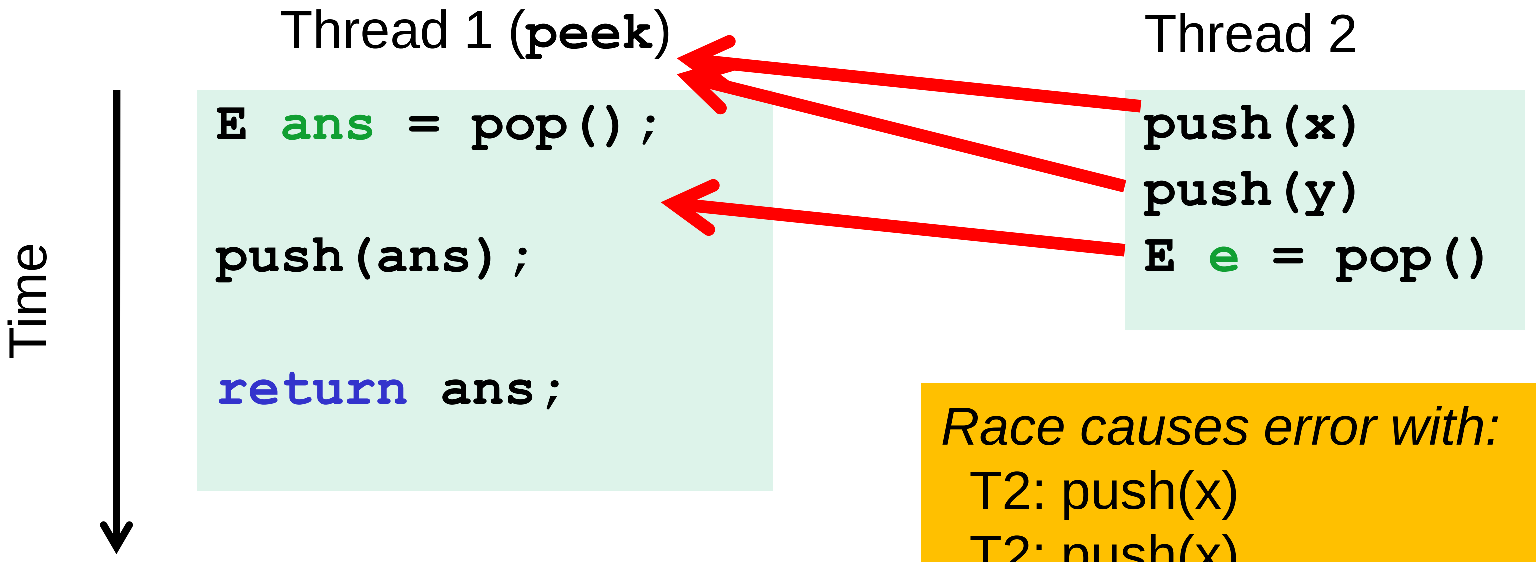
Race causes error with:
T2: push(x)
T1: pop()
T2: push(x)
T1: push(x)

Example 2: peek and push

- **Property we want:**

  Values are returned from **`pop`** in LIFO order

- With **`peek`** as written, property can be violated – how?



Thread 1 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

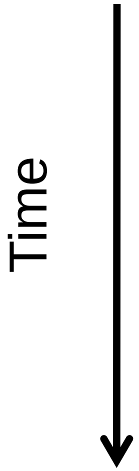
```
push(x)
push(y)
E e = pop()
```

Race causes error with:
T2: push(x)
T2: push(x)
T1: pop()
T2: pop()

Example 3: peek and peek

- **Property we want:**
  `peek` does not throw an exception unless stack is empty

- With **`peek`** as written, property can be violated – how?



Thread 1 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

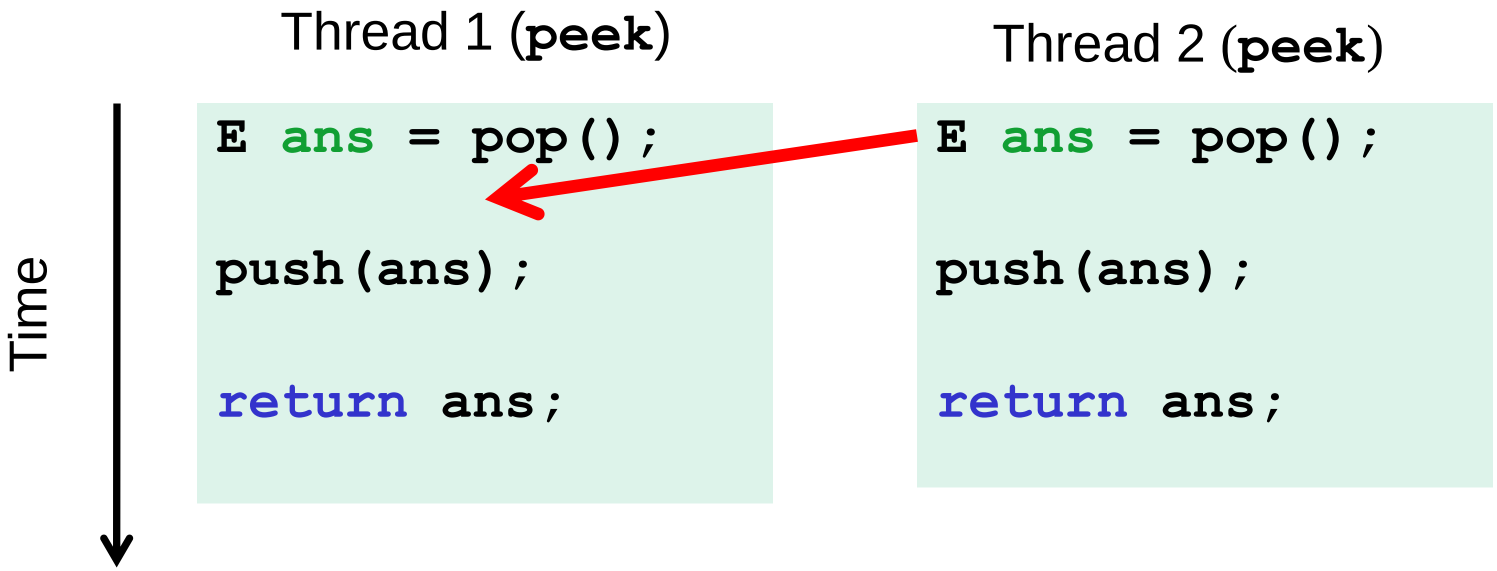
Thread 2 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

Example 3: peek and peek

- **Property we want:**
  `peek` does not throw an exception unless stack is empty

- With **`peek`** as written, property can be violated – how?



Thread 1 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

The Fix

- In short, **`peek`** needs synchronization to disallow interleavings
  - The key is to make a *larger critical section*
    - This protects the intermediate state of `peek`
  - Use re-entrant locks; will allow calls to **`push`** and **`pop`**
  - Can be done in stack (on left) or an external class (on right)

```
class Stack<E> {
  …
  synchronized E peek(){
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

```
class C {
  <E> E myPeek(Stack<E> s){
    synchronized (s) {
      E ans = s.pop();
      s.push(ans);
      return ans;
    }
  }
}
```

An Incorrect "Fix"

- So far we have focused on problems created when **`peek`** performs **writes** that lead to an incorrect intermediate state

- A tempting but incorrect perspective
  - If an implementation of **`peek`** does not write anything, then maybe we can skip the synchronization?

- Does not work due to *data races* with **`push`** and **`pop`**
  - Same issue applies with other readers, such as **`isEmpty`**

Another Incorrect Example

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  boolean isEmpty() { // unsynchronized: wrong?!
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    return array[index--];
  }
  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

Why Wrong?

- It *looks like* **`isEmpty`** and **`peek`** can “get away with this” because **`push`** and **`pop`** adjust the state “in one tiny step”

- But this code is still *wrong* and depends on language-implementation details you cannot assume
  - Even “tiny steps” may require multiple steps in implementation: **`array[++index] = val`** probably takes at least two steps
  - Code has a data race, allowing very strange behavior

- Do not introduce a data race, even if every interleaving you can think of is correct

Getting it Right

Avoiding race conditions on shared resources is difficult

- Decades of bugs have led to some *conventional wisdom*, general techniques that are known to work

Rest of lecture distills key ideas and trade-offs

- More available in the suggested additional readings
- But none of this is specific to Java or a particular book
  - May be hard to appreciate in beginning
  - Come back to these guidelines over the years
  - Do not try to be fancy