Threads and Concurrency in Java: Part 2

Waiting

- Synchronized methods introduce one kind of coordination between threads.
- Sometimes we need a thread to wait until a specific condition has arisen.

Waiting by "Polling"

- For example in a Counter class we might want to wait until the count is 0 or less.
- We could do this

```
class Counter {
    private int count ;

    public Counter( int count ) { this.count = count ; }

    synchronized void decrement() { count -= 1; }

    synchronized int getCount() { return count ; }

    void waitForZeroOrLess() { // Polling loop
        while( getCount() > 0 ) /*do nothing*/;
    }
}
```

Note that waitForZeroOrLess is not synchronized

Waiting by "Polling"

- Slightly improving this we can write

```
void waitForZeroOrLess() { // Polling loop
    while( getCount() > 0 )
        Thread.yield();
}
```

which keeps the thread from hogging the CPU while polling.

Waiting by *wait* and *notifyAll*

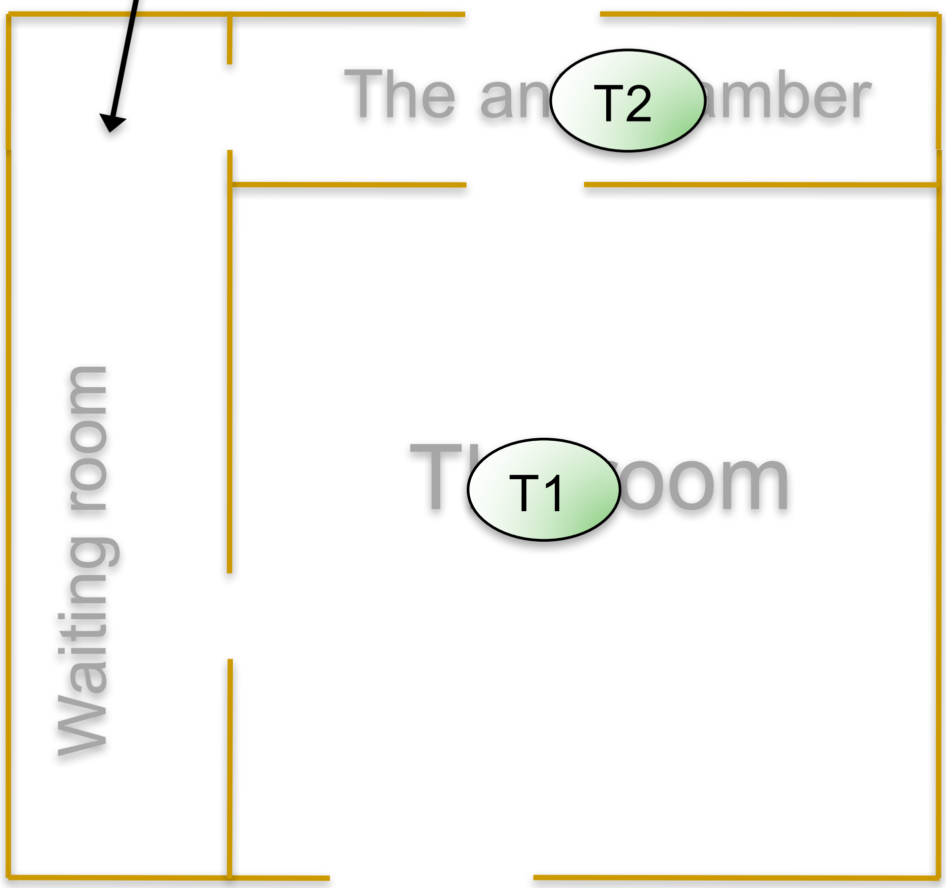
- Better, we can have a thread wait until it is notified that the condition has (or may have) come about.
- While waiting, the thread relinquishes ownership and waits until some later time.
- To wait, a thread sends the object a *wait* message.
- To allow other threads to stop waiting, threads sends a *notifyAll* message to the object.
- *wait* and *notifyAll* are methods of class Object.

The "Room metaphor"

Add a waiting room

When a thread in the room sends a *wait* message, it leaves the room and enters the waiting room.

Later another thread enters.



It sends a *notifyAll* message.

All threads in the waiting room move to the antechamber.

And so they can reenter when the room is empty.

wait and *notifyAll* example.

Note that waitForZeroOrLess ***is*** **synchronized**

```
class Counter {
        private int count ;

        public Counter( int count ) { this.count = count ; }

        synchronized void decrement() {
                count -= 1 ;
                if( count <= 0 ) notifyAll() ;
        }

        synchronized void waitForZero() {
                while( count > 0 ) {
                        try { wait(); }
                        catch( InterruptedException e ) {}
                }
        }
}
```

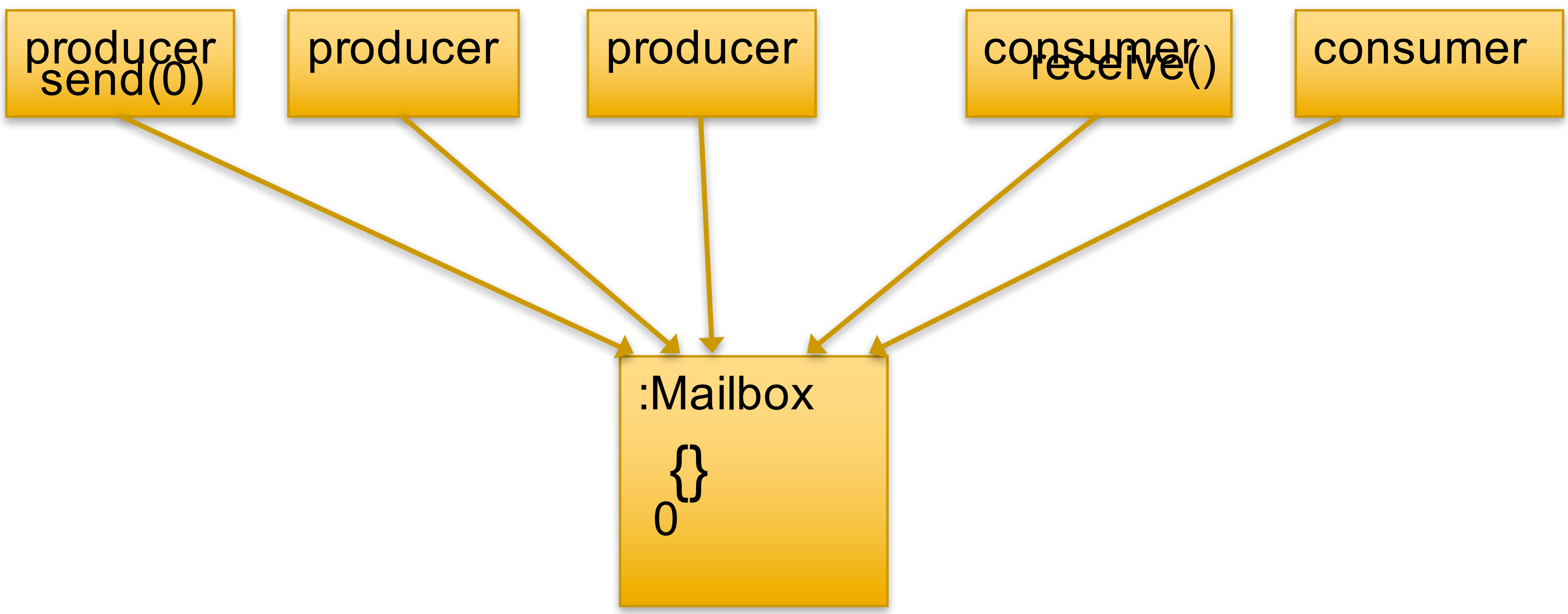
A possible sequence

- One thread
  - Calls waitForZero enters the room
  - Sees count is 1.
  - calls *wait*, enters the waiting room
  - waits in waiting room
  - …
  - …
  - …
  - …
  - waits in antechamber
  - …
  - …
  - reenters the room
  - returns from wait
  - sees count is 0 and returns
- Another thread

  Calls decrement enters the room.
  - count--
  - call *notifyAl,l* moving other thread to wait queue for the lock
  - returns leaving the room

Passing messages

- We want to pass messages between any number of producers and consumers executing on any number of threads.

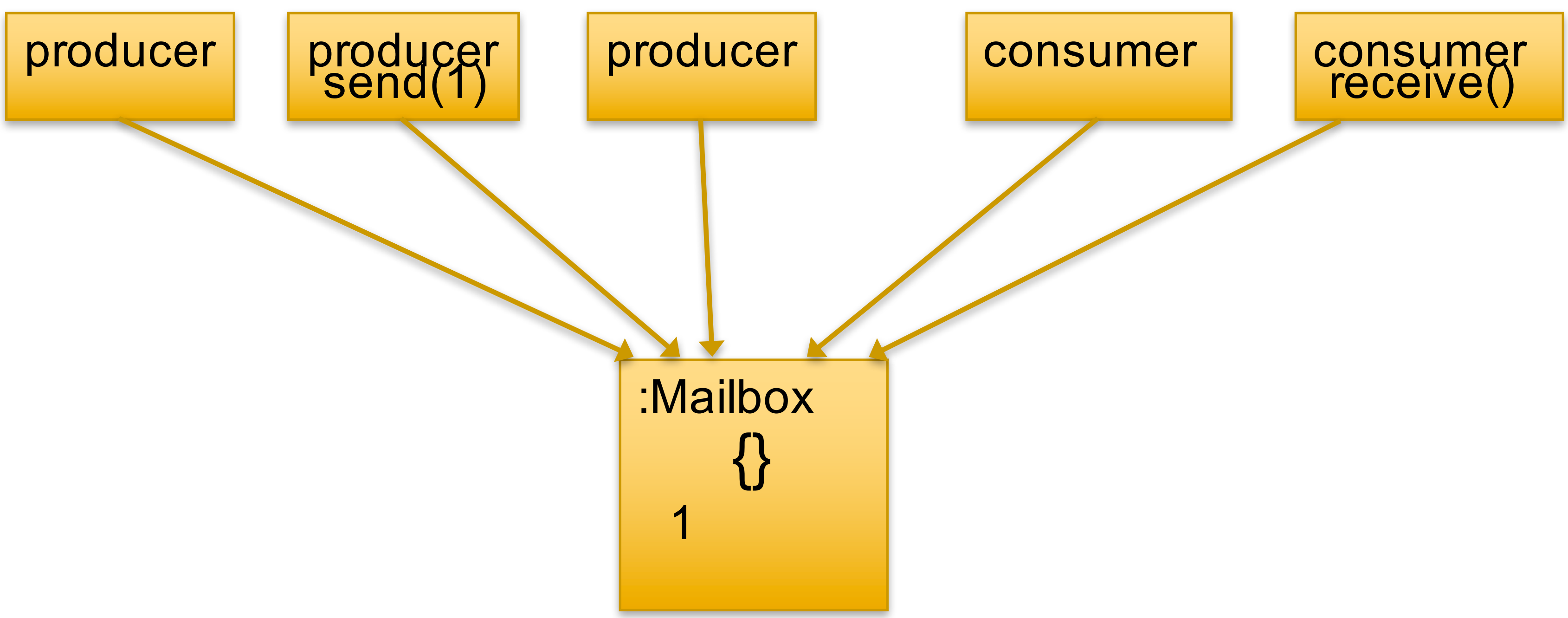


Passing messages

- We want to pass messages between threads so that
  - Each message is received no more than once
  - No message is overwritten before it is received.
  - A receiver may have to wait until a new message is sent
  - A sender may have to wait until there is room in the queue
  - Up to 100 messages can be sent but not yet received.

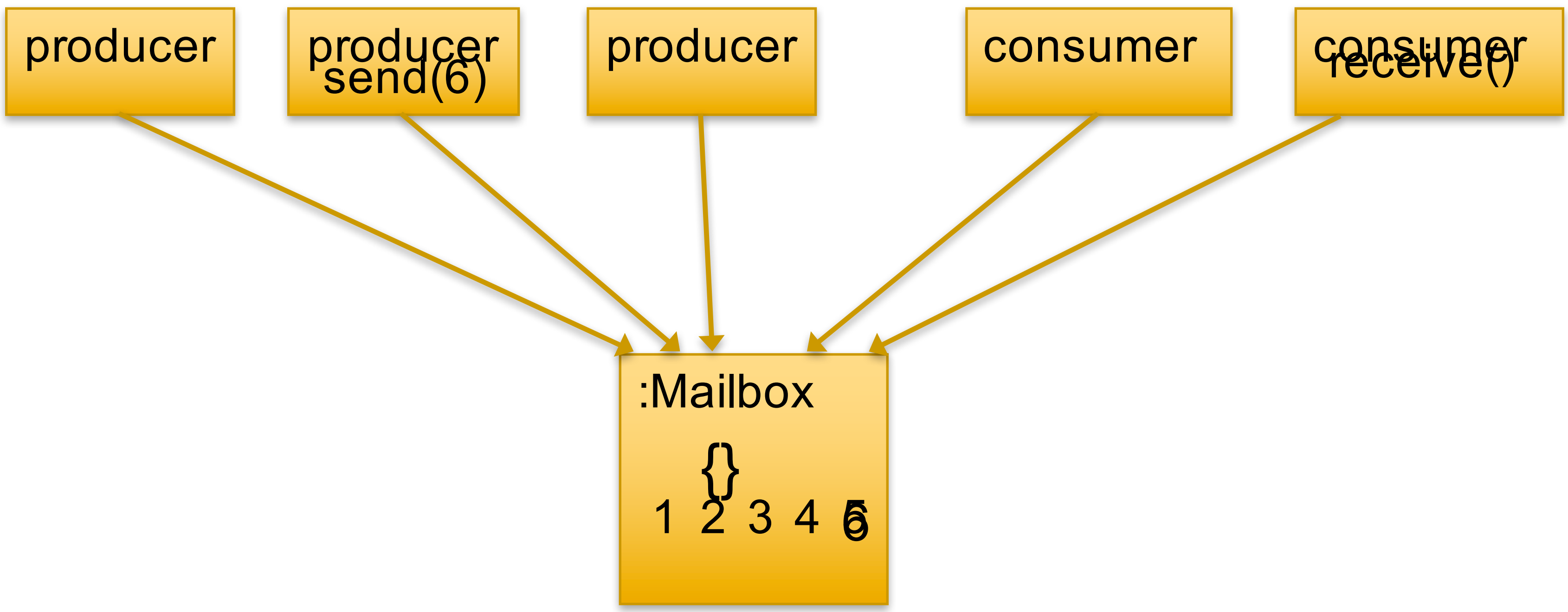
Passing messages

- Receivers may need to wait.
  - Suppose the mailbox initially has no messages



Passing messages

- Producers may need to wait.
  - Suppose the capacity is 5 and the mailbox is full



Passing messages – incomplete

```
class Mailbox<MessageType> {
    private static final int CAP = 100 ;
    private Queue<MessageType> q = new Queue<MessageType>() ;
    // invariant: q.size() <= CAP

   public synchronized void send(MessageType mess) {
        // wait until q.size()< CAP                    To do
        q.put( mess );


   }
   public synchronized MessageType receive() {
        // wait until q.size() > 0                     To do
        return q.take();


   }
```

Passing messages – add waits

```
class Mailbox<MessageType> {
    private static final int CAP = 100 ;
    private Queue<MessageType> q = new Queue<MessageType>() ;
    // invariant: q.size() <= CAP

    public synchronized void send(MessageType mess) {
        // wait until q.size()< CAP
        while( q.size()==CAP ) {
            try { wait() ; } catch( InterruptedException e ) {}
        }
        q.put( mess )
    }
    public synchronized MessageType receive() {
        // wait until q.size() > 0
        while( q.size()==0 ) {
            try { wait() ; } catch( InterruptedException e ) {}
        }
        return q.take()
    }
}
```

To do

To do

Passing messages – add notifications

```
class Mailbox<MessageType>{
    private static final int CAP = 100 ;
    private Queue<MessageType> q = new Queue<MessageType>() ;
    // invariant: q.size() <= CAP

    public synchronized void send(MessageType mess) {
        // wait until q.size()< CAP
        while( q.size()==CAP ) {
            try { wait() ; } catch( InterruptedException e ) {}
        }
        q.put( mess );   notifyAll() ;
    }
    public synchronized MessageType receive() {
        // wait until q.size() > 0
        while( q.size()==0 ) {
            try { wait() ; } catch( InterruptedException e ) {}
        }
        notifyAll() ;
        return q.take();
    }
```

Done

Even better than wait and notifyAll

- As software gets more complex, using “wait” and “notifyAll” can be a bit awkward and is easy to mess up.
- An improvement is to use Dr. Norvell’s “monitor” package.
- See http://www.engr.mun.ca/~theo/Misc/monitors/monitors.html

Deadlock

- While waiting can solve “safety” issues
  - (e.g. ensuring noncorruption of data, nondup-lication of messages, nonloss of messages),
- it can cause “liveness” problems.
- In particular
  - if one thread is waiting for another to do something, and
  - the other thread is waiting for the first to do something,
  - then we have “deadlock”

Deadlock Example

- Suppose we have a bank account class

```
class Account {
    private int balance = 0.0;

    public synchronized void addFunds( int amount ) {
        balance += amount;
    }
    public void transfer ( int amount, Account toAccount ) {
        if( balance >= amount ) {
            toAccount.addFunds( amount );
            balance -= amount;
        } else { … } }
    …
}
```

Deadlock Example

- There is a subtle problem here.
- The intent is that one should not be able to transfer out of an account more money than it has. (A safety problem.)
- But, if two threads attempt to transfer from the same account at about the same time, then they might both succeed, even though the final balance will be negative.
- To fix this, we make transfer **synchronized**.

Deadlock Example

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

    public synchronized void addFunds( int amount ) {
        balance += amount;
    }
    public synchronized void transfer (int amount, Account toAccount){
        if( balance >= amount ) {
            toAccount.addFunds( amount );
            balance -= amount;
        } else { … } }
    …
}
```

Deadlock Example

- But now deadlock is possible.
- Suppose thread 0 tries to transfer from account x to account y.
- At roughly the same time thread 1 attempts to transfer from account y to account x

A possible sequence

- Thread 0
  - calls x.transfer(100, y)
  - obtains a lock on x
  - calls y.addFunds()
  - waits for lock on y
  - waits for lock on y
  - waits for lock on y
  - waits for lock on y
  - ad infinitum
- Thread 1
  - calls y.transfer(50, x)
  - obtains a lock on y
  - calls x.addFunds()
  - waits for lock on x
  - waits for lock on x
  - waits for lock on x
  - waits for lock on x
  - ad infinitum

The Threads are now deadlocked!

A solution to deadlock

- One solution is to always lock objects in a particular order.
- e.g. give each lockable object a globally unique #
- The following example uses synchronized blocks

```
public void transfer ( int amount, Account toAccount ) {
    boolean choice =  this.serialNum() <= toAccount. serialNum() ;
    synchronized( choice ? this : toAccount ) {
       synchronized( choice ? toAccount : this ) {
          if( balance >= amount ) {
              toAccount.addFunds( amount ) ;
              balance -= amount;
          } else { … } } } }
}
```

Testing Concurrent Programs

- You can not effectively test concurrent programs to show that they are error free
- Because of race conditions, a test may pass millions of times “by chance”.
- Tests that fail are useful. *They tell us we have a bug.*
- Tests that pass only tell us that it is *possible* for the code to compute the correct result.

Testing: A True Story

- I wanted to illustrate how race conditions can cause data corruption.
- So I wrote a program with two threads sharing an int variable x.
  - Initially x was set to 0
  - One thread incremented x a thousand times
  - The other thread decremented x a thousand times.

Testing: A True Story

```
class Incrementor extends Thread {
    public void run() {
        for(int i=0 ; i < 1000; ++i) ++x ; }
}
class Decrementor extends Thread {
    public void run() {
        for(int i=0 ; i < 1000; ++i) --x ; }
}
```

Testing: A True Story

- I ran the two threads concurrently

```
System.out.println( "The initial value of x is: " + x ) ;

Thread p = new Incrementor() ;
Thread q = new Decrementor() ;
p.start() ; q.start();

// Wait for threads to finish using "joins" (not shown)

System.out.println( "After "+1000+" increments and "+1000+"
    decrements" ) ;
System.out.println( "the final value of x is: " + x ) ;
```

- What do you think happened?

Testing: A True Story

- Here's the output:

  The initial value of x is: 0

  After 1000 increments and 1000 decrements

  the final value of x is: 0

- Even though I deliberately wrote a faulty program and gave it 1 thousand chances to fail the test, it passed anyway..
  - I tried 10,000. Then 100,000.
  - Same result

Testing: A True Story

- And why did it pass?
  - The JVM happened to give each thread time slices so long that the incrementing thread completed its 100,000 increments before the main thread had a chance to even start the decrementing thread.
- Changing to 1,000,000 increments and decrements revealed the bug.

Testing: A True Story

- I had fallen victim to <u>optimism</u>.
- I had optimistically assumed that such an *obvious* bug would cause a *thorough* test to fail.
- If tests designed to reveal *blatant* bugs, *which we know are in the program*, fail to reveal them, should we expect testing to reliably reveal subtle bugs we do not know about?

If you can't test, then what?

- The good news is that
  - although testing is insufficient to reveal bugs,
  - there are design and analysis techniques that allow you to prove your programs correct.

Concurrency

- What every computer engineer needs to know about concurrency:

  Concurrency is to untrained programmers

  as matches are to small children.

  It is all too easy to get burned.

  - Race conditions
  - Deadlock
  - Insufficiency of testing

Summary of terminology

- **concurrency:** multiple agents running at the same time, interacting
- **thread**: an independent path of control
- ***Thread***: a Java class representing threads
- **race condition**: a hazard caused by the unpredictability of execution timing
- **synchronized access**: locking of objects to obtain **exclusive access**
- **wait** and **notifyAll**: threads may wait until notified
- **deadlock**: a cycle of threads mutually waiting for each other
- **safety property**: a property that says something (bad) will never happen
- **liveness property**: a property that says something (good) will eventually happen