Exceptions and Threads

Exceptions

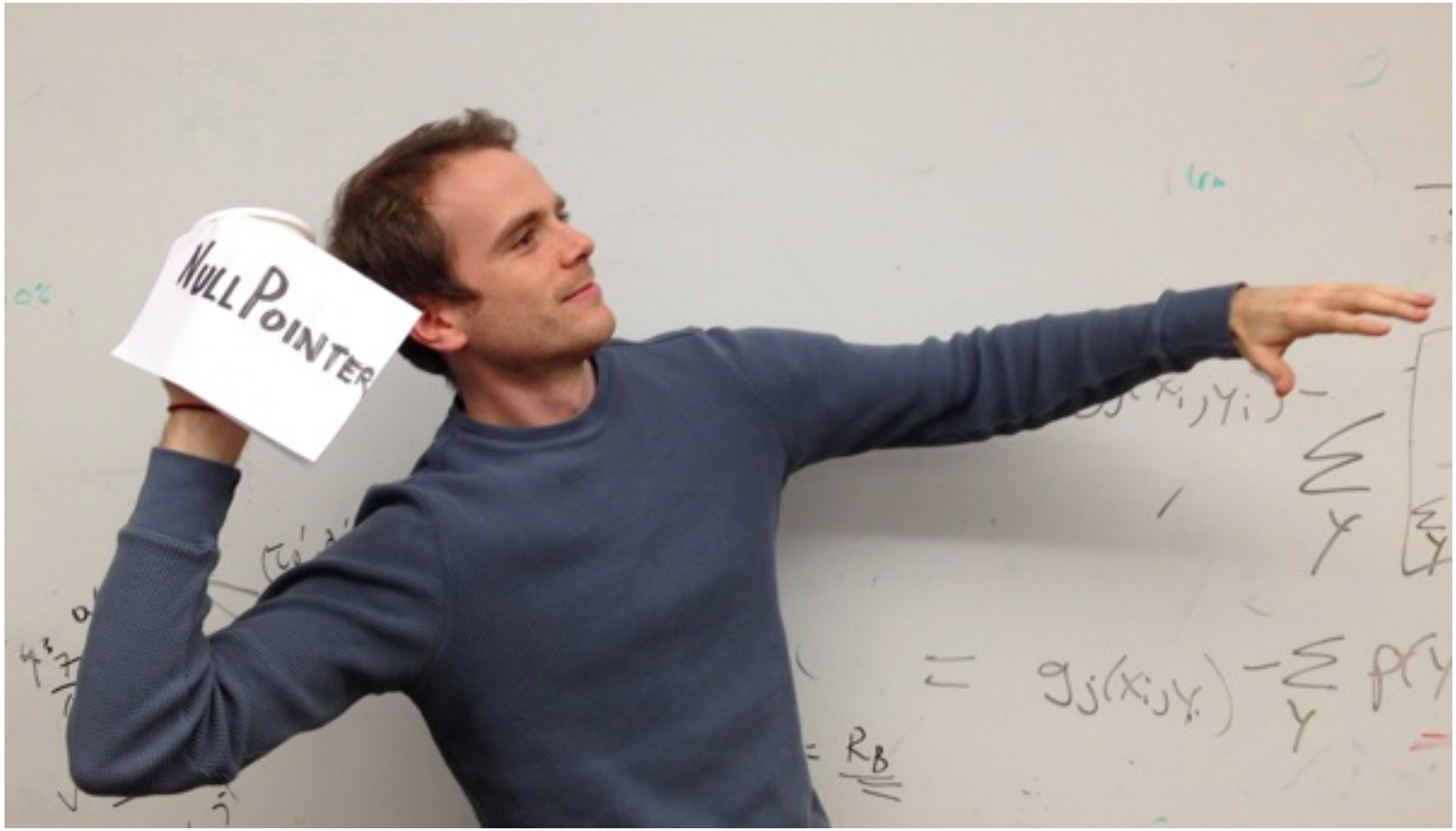
- What do you do when a program encounters an anomalous, unusual event?
  - Try to open a file and it's not there
  - Try to convert a string to an integer and it's not a valid integer
  - Try to dereference a pointer and it's null

Exceptions

- You could
  - crash the program
    - Not a great idea
  - return an error code
    - But what if all return values are "meaningful?"
  - force the user to manually check the condition before taking the action that might cause problems
    - More work for the programmer

Exceptions

- Java (and other languages) choose to "throw an exception."



Exceptions

- An *Exception* is an encapsulation of a problem that occurred while your program was running.
- Exceptions allow the programmer to separate the logic of the exceptional situation itself from what to do about it.
  - The other ways usually force you to couple together the code that generated the error with the code that handles the error situation.

Exceptions

- When an exceptional situation occurs, your code can choose to "throw an exception."
- When this happens, another piece of code must "catch the exception."

```
try {
  Scanner sc = new Scanner(new File("data.txt"));
  // read data from the scanner…
}
catch (FileNotFoundException e) {
  System.err.println("Couldn't open file.");
}
```

- Any code that has the ability to throw an exception should be placed inside a try block.
  - Here, the Scanner constructor may throw an exception if it can't find data.txt
- The catch block afterwards is the error handler code.

```
try {
  Scanner sc = new Scanner(new File("data.txt"));
  // read data from the scanner…
}
catch (FileNotFoundException e) {
  System.err.println("Couldn't open file.");
}
```

- If the code in the try block *doesn't* throw an exception, the catch block is skipped.
- If the code in the try block *does* throw an exception, as soon as the exception happens, the catch block starts running. After it finishes, program continues with whatever is after the catch block.
  - Therefore you can recover from errors gracefully.
  - Error handling logic is separated from the "normal program" logic.

- Methods that have the ability to throw exceptions must declare what exceptions are possible.

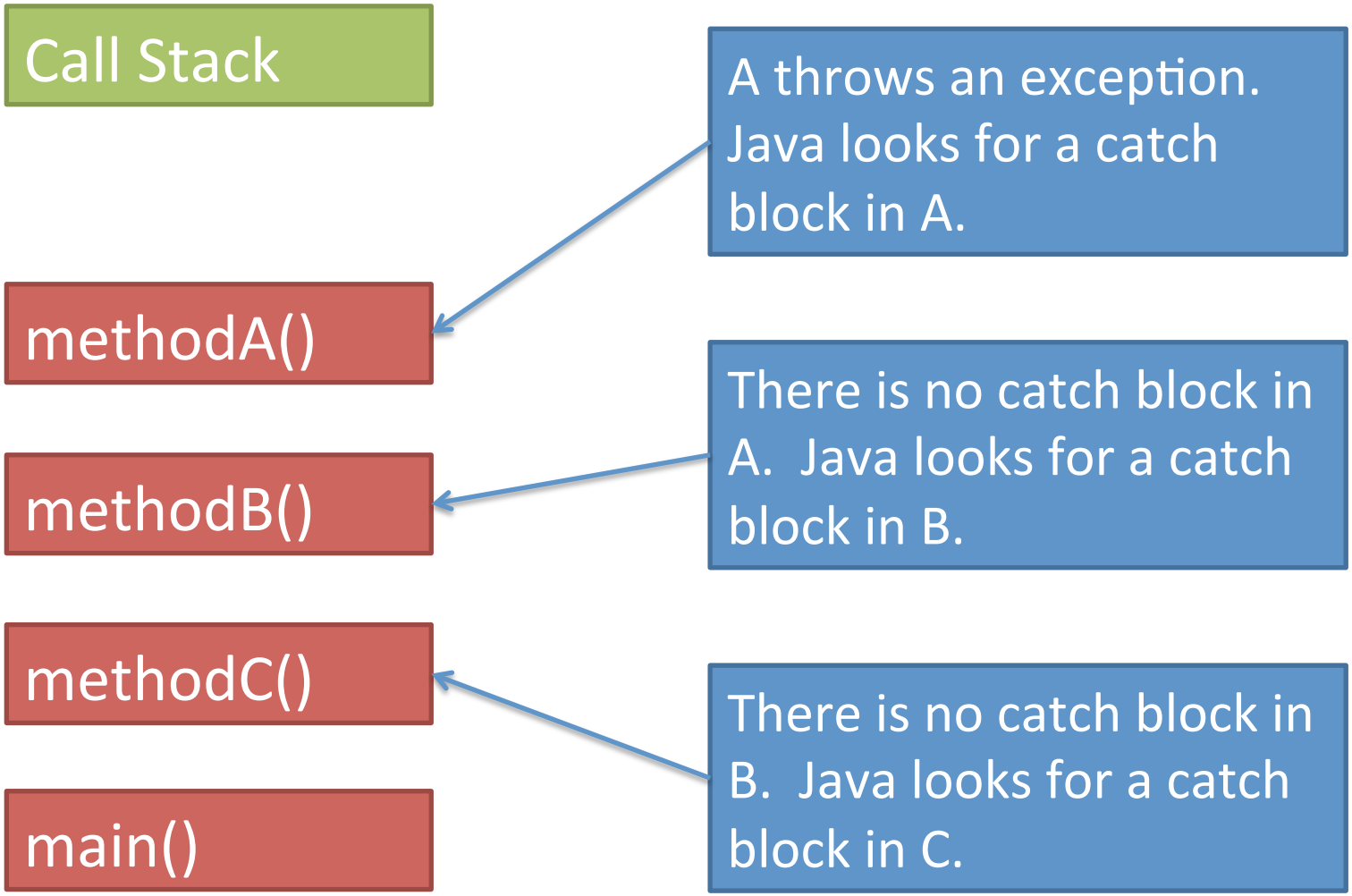
```
public Scanner(File source)
    throws FileNotFoundException {
        ...
}
```

- Java API tells you which methods throw which exceptions.
- Code will not compile without proper try/ catch blocks.

Code can further decouple the "throwing" logic from the "catching" logic:

```
void methodA() throws SomeException {
   // code here that may throw SomeException
}
void methodB() throws SomeException {
   methodA()
}
void methodC() {
   try { methodB(); }
   catch (SomeException e)
    { … }
```

If a method wants to call some code that may throw an exception, the method must either handle it (with a catch block) or pass it back to the calling method (add "throws" to the declaration line).


Call Stack
methodA()
methodB()
methodC()
main()
A throws an exception. Java looks for a catch block in A.
There is no catch block in A. Java looks for a catch block in B.
There is no catch block in B. Java looks for a catch block in C.

- "Normal" Exceptions
  - Inherit from class **`Exception`**. Must be caught with a try block somewhere.
- Runtime Exceptions
  - Inherit from class **`RuntimeException`**. Do not have to be caught.
  - DivideByZeroException, IndexOutOfboundsException, NullPointerException.
- Errors
  - Inherit from class **`Error`**. Do not have to be caught because they indicate something a reasonable application probably can't recover from anyway (e.g., out of memory, stack overflow).

Takeaway

- There are some methods that force you to write error-handling code. Won't compile without the try-catch.
- Wrap the error-causing code in a try block (can wrap as much code as you want), and then put a catch block and try to do something intelligent in it (can be as simple as printing a message.)

More advanced stuff

- Writing your own Exception classes
- Writing your own methods that throw Exceptions (you can also throw exceptions that come with Java)
- Beyond the scope of this class; consult a Java book; won't be necessary for projects or exams.
- C++ also has exceptions; other languages too.

Threads

- Most programs you write do one thing at a time.
- Execution proceeds in a linear fashion, where the previous command always completes before the next one starts.
- Sometimes we need to write programs that do multiple things at once.

- Examples
  - Display a loading animation while accessing a big file.
    - e.g., web browsers
  - Handling requests in a client-server application.
    - e.g., web servers
  - Monitoring some situation in the background while letting the program do other things.
    - e.g., your email application
  - Games, games, games (and other GUI stuff)
    - Separate threads to handle information coming from keyboard, mouse, network.

- A single CPU really can't do multiple things at once.
  - If you have multiple CPUs, OK.
- Simulated by switching back and forth between tasks really quickly.

Processes vs threads

- A *process* is a self-contained execution environment.
  - Process is often synonymous with "program" or "application" but not always.
  - Most importantly, each process has its own memory space.
  - Processes can communicate with each other through interprocess communication (IPC) [see networking class]

Processes vs threads

- A *thread* is an execution environment within a process.
  - Within a process, there can be multiple threads, and they all share the same memory space.
  - Threads communicate with each other through variables (memory is shared, so variable are shared among threads).
- By default, all programs are single-threaded.
  - These are the kinds of programs you've been writing so far.

Java Threads

- Every thread is associated with a `Thread` object.
- The Thread class has a single method that you will override:

```
public void run()
```

- The code inside this method defines what the thread will do.
- To start the thread, call the `start()` method.
  - You never directly call `run()` yourself.

- ThreadEx1, ThreadEx2, CountingEx

Takeaway

- A call to start() ***returns immediately***.
- The code in run() then starts running in a thread parallel to your main program.

rest of main()	t1's run()	t2's run()
print message that both threads have started	print 0 print 1 print 2 print 3 …	print 0 print 1 print 2 print 3 …

Sleeping

- Threads can go to sleep, which pauses that thread for a certain amount of time.
- During that time, the CPU will only deal with other threads.
- After the time is elapsed, the thread wakes up and continues.

Good sleep

```
System.out.println("Falling asleep!")
try
{
  // goes to sleep for one second
  Thread.sleep(1000)
} catch (InterruptedException e) { }

System.out.println("Now I'm awake!")
```

Bad sleep

```
int start = System.currentTimeMillis()
int finish = start + 1000;
while (System.currentTimeMillis() < finish)
{
}
```

InterruptedException

- Some thread methods throw InterruptedException, which must be caught.
- You can decide what to do with it.
- Fine to ignore it (for this course).

Join

- Also common to want to pause execution of a thread until another thread finishes.
- If t is a thread object, you can call

  `t.join()`

  This will pause the current thread (*a la* **`sleep()`**) but will wake up as soon as t finishes.
- Example: CountingJoin

- So far, threads are easy!

- So far, threads are easy!

- Where threads become hard is when they start sharing variables.

- Imagine two ATMs and two people who have a shared account. The account has $20.
- Both people go up to two different ATMs at the same time. Both try to withdraw $20 simultaneously.

```
void withdraw(int amount) {
   if (balance >= amount)
      balance -= amount;
}
```

balance >= amount has multiple steps:

- Retrieve the current value of balance.
- Retrieve the current value of amount.
- Compare those two values.

balance >= amount has multiple steps:

- Retrieve the current value of balance.
- Retrieve the current value of amount.
- Compare those two values.

ATM 1: Retrieve current balance (= 20)
ATM 2: Retrieve current balance (= 20)
ATM 1: Retrieve current amount (= 20)
ATM 2: Retrieve current amount (= 20)
ATM 1: Compare => true
ATM 2: Compare => true

Both ATMs dispense cash!

- So it appears we can withdraw $40 from a $20 balance!
- And then our balance would be negative!
- But no, it's much, much worse.

balance -= amount has multiple steps:

- Retrieve the current value of balance.
- Retrieve the current value of amount.
- Subtract, put result in balance.

balance -= amount has multiple steps:

- Retrieve the current value of balance.
- Retrieve the current value of amount.
- Compare those two values.

ATM 1: Retrieve current balance (= 20)
ATM 2: Retrieve current balance (= 20)
ATM 1: Retrieve current amount (= 20)
ATM 2: Retrieve current amount (= 20)
ATM 1: Subtract => 0 => store 0 in balance
ATM 2: Subtract => 0 => store 0 in balance
Both ATMs dispense cash!

- Pathological example; very possible that nothing bad will happen at all.
  - And then you don't notice anything bad happening until your bank starts mysteriously losing money ever so often…

- Called a ***race condition***.
  - Race condition: situation where result is dependent on the sequence or timing of other, uncontrollable events.
  - This specific condition is a *memory inconsistency error*: Happens when different threads have inconsistent views of what should be the same information.

Demo

- Ex: Bank
- Ex: Bank2 (race)

Solution: locks

- Every object has a *lock* associated with it.
  - Sometimes called an intrinsic lock or monitor lock.
  - Note: separate locks for each instance!
- A lock can be owned by at most one thread.
  - Sometimes owned by no threads.
- Can be used to prevent race conditions by forcing methods to own the object's lock before running code that needs exclusive access to that object's fields.

Locks

- Locks are not objects themselves.
- Access to them is controlled through blocks of code that are declared as "synchronized."

- When a thread T1 attempts to enter a block of code that is synchronized on object x, T1 tries to acquire x's lock.
  - If x's lock is available, then T1 acquires the lock and runs the block of code.
  - If x's lock is not available (owned by another thread), then the scheduler switches to a different thread. At some point, the scheduler will switch back to T1 and try again to acquire the lock.
- When T1 leaves the synchronized block, x's lock is released.

- First kind of synch block: *synchronized method.*

- Use the word **`synchronized`** after the return type in the declaration line of a method.
- If `method()` is synchronized, when a thread calls `x.method()`, the thread will try to acquire x's lock.

```
Class C {
  synchronized void methodA() { }
  synchronized void methodB() { }
}
in main:
  C x = new C(), y = new C();
  // two threads start simultaneously
```

Thread 1:	Thread 2:
x.methodA() // 1 acquires x's lock. // 1 starts running methodA // 1 finishes methodA // 1 releases x's lock	x.methodA() // 2 fails to acquire x's lock // 2 acquires x's lock // 2 starts running methodA // 2 finishes methodA // 2 releases x's lock

```
Class C {
  synchronized void methodA() { }
  synchronized void methodB() { }
}
in main:
  C x = new C(), y = new C();
  // two threads start simultaneously
```

Thread 1:	Thread 2:
x.methodA() // 1 acquires x's lock. // 1 starts running methodA // 1 finishes methodA // 1 releases x's lock	x.methodB() // 2 fails to acquire x's lock // 2 acquires x's lock // 2 starts running methodB // 2 finishes methodB // 2 releases x's lock

```
Class C {
  synchronized void methodA() { }
  synchronized void methodB() { }
}
in main:
  C x = new C(), y = new C();
  // two threads start simultaneously
```

```
Thread 1:

x.methodA()
  // 1 acquires x's lock.
  // 1 starts running methodA
  // 1 finishes methodA
  // 1 releases x's lock
```

```
Thread 2:

y.methodA()
  // 2 acquires y's lock.
  // 2 starts running methodA
  // 2 finishes methodA
  // 2 releases y's lock
```

- If T1 owns x's lock, (presumably because T1 has already synchronized on x), T1 may enter another synchronized method of x.
- In other words, if you try to acquire a lock you already own, nothing bad happens.
  - Happens when synch blocks call other functions that have synch blocks.

- CPU can still stop a thread T1 in the middle of a synch block and switch to a different thread T2.
  - In other words, synchronized doesn't mean the method can't get interrupted in the middle by another thread.
- If T2 happens to need a lock owned by T1, then the scheduler will immediately switch again.

Fix bank account

Fix the race condition in bank2.

- Also can have synchronized blocks (inside any method):

```
class C {
  public void method() {
    synchronized (y) {  …  }
  }
in main: C x = new C(); x.method()
```

- When a thread tries to call `x.method()`, the thread will try to acquire the lock for some other object y, not x.

Fix bank account

Fix the race condition in bank2 in a different way.

Assume we have five Fork instances.

Inside each philosopher's run method:

```
synchronized (fork to the left) {
  synchronized (fork to the right) {
    // eat spaghetti
  }
}
```

Deadlock

Remedies

- Resource hierarchy: assign numbers to the forks; can't request a higher-numbered for before a lower-numbered fork.
- Central arbiter: Write a waiter class that manages all the forks. The waiter will never give out forks in a way that will allow deadlock.

Other issues

- Starvation
  - A thread is consistently denied access to a shared resource by other "greedy" threads.
  - Example: synch methods that take a long time to run and are called frequently.
- Livelock
  - Thread A takes some action in response to another Thread B in attempt to avoid a problem.
  - Thread B then response to A's action.
  - Back and forth: neither thread is deadlocked, but they are too busy responding to each other to get anything else done.

Coordination

- Imagine a restaurant with a chef and a waiter.
- The chef's job is to prepare food and place the food in the pickup area.
- The pickup area can only hold one order at a time.
- The waiter's job is to take the food from the pickup area to the tables.

- Class PickupArea models the waiting area for an order. Holds the order number as an int.
- Class Chef is a thread that when started, will cook ten orders back to back (sleeping randomly between them) and place them in the waiting area.
- Class Waiter is a thread that when started, will pick up ten orders from the waiting area and serve them (sleeping randomly between them).

- Show restaurant

- Waiter doesn't wait for chef to cook meals before serving them.
  - The waiter might serve the same meal over and over, or sometimes will serve order 0, which means there is no meal!
- Chef doesn't wait for the pickup area to be empty before cooking the next meal.
  - The chef might cook multiple orders and put them all in the waiting area back to back, overwriting the existing order that was already there.

2 part solution

- Part A:
  - Synchronize on the pickup area so that the waiter and chef don't step on each other's toes.
- Part B:
  - Have the two threads communicate about when orders are ready.

Solution: Guarded blocks

- A guarded block is a block of code that cannot execute until a condition is true.
- Chef should not cook a new order until the pickup area is free.
- Waiter should not pickup an order unless there is one waiting in the pickup area.

In Chef.run():

```
while (pickupArea.orderNumber > 0) { }
```

In Waiter.run():

```
while (pickupArea.orderNumber == 0) { }
```

Let's try.

Busy waiting is bad, mm'kay?

- Never wait on a condition with an empty while loop.
- If a thread cannot continue until a condition is true, we need to tell the thread to wait without wasting CPU cycles.

- Every object has two methods, called wait() and notifyAll()
- Inside a synchronized block on object x, a thread may call wait() and/or notifyAll()
- x.wait() suspends the current thread until it receives a wakeup call from x.notifyAll()
- x.notifyAll() wakes up all the threads that are waiting on object x.

Most common idiom:

T1: [inside synch for x]

```
while (!condition) { x.wait(); }
```

T2:

```
condition = true; x.notifyAll();
```

Try it out

Why does this work?

- If T1 holds x's lock and calls x.wait(), then x's lock is temporarily released!
- Therefore, another thread T2 can acquire x's lock to fix the condition that T1 is waiting on.
- Busy waits and sleep()s don't release locks, so our first fix just got stuck forever waiting.

Bank account vs Restaurant

- BankAccount worked with synchronized methods only because if we try to withdraw more money than we have, the withdraw() method simply ***fails***.

Bank account vs Restaurant

- Chef & Waiter needs wait/notifyAll because:
  - We don't want the Chef to lose an order (fail) if there's already an order waiting to be picked up (aka when the Chef is ahead of the Waiter)
  - We don't want the Waiter to pick up the same order twice (fail) if there's not a new order waiting to be picked up (aka when the Waiter is ahead of the Chef).