CS 1181
Week Ten

Reese Hatfield

Searching

Reese Hatfield

Searching

- Ever used ArrayList.indexOf(Element)?
- Useful method when:
  - You need to check if the value is present
  - Where that value actually is

Searching

- IndexOf essentially just searches the list to find where the element is

- How could we implement this ourselves?

Searching

- Well you would just iterate thru it right?

- Often times, this is the best approach
- Let’s do it, but:
  - Let’s count the #operations
  - Operation = a check if cur == target

Searching

- This approach is often called a “Linear Search”

- If your list has N elements, it will take you time proportional to N to find your target
- O(N) time complexity

Searching

- Is this *really* the best way?
- If I gave you a dictionary, and said lookup the definition of “University”
- What would you do?

Searching

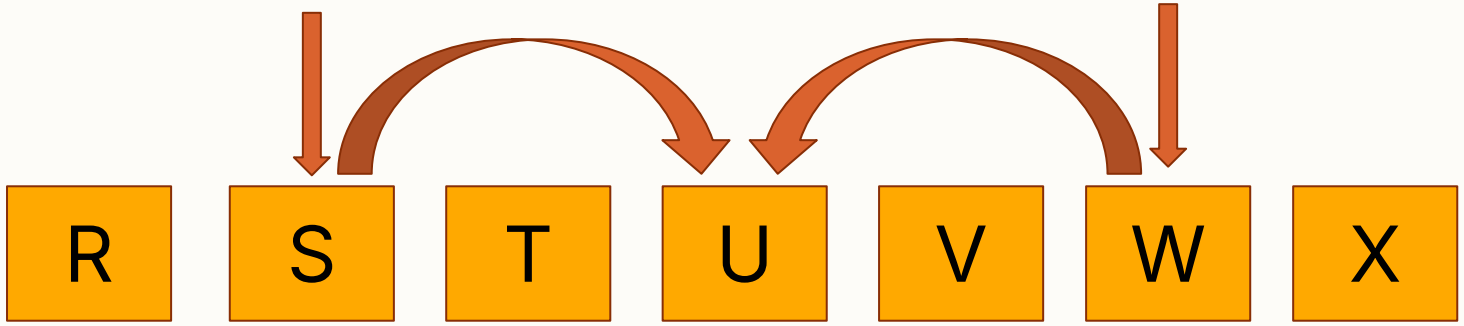
- What you're *not* going to do is start on page 1
  - Aardvark
  - Above
  - Abyss
  - ...
  - University

Searching

- That would take forever
- You’re probably gonna start somewhere in the middle
- Adjust according to wherever you happened to land

Searching

- If you landed at
  - S ⇒ go right
  - W ⇒ go left

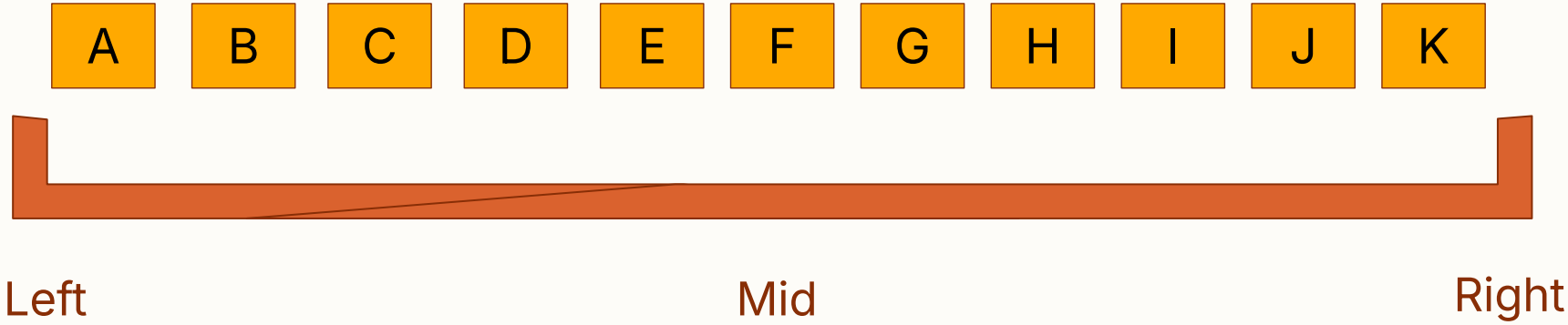


Searching

- We are only able to do this since we *know* the list is in alphabetical order
- To code something like this
  - We can narrow a sliding window

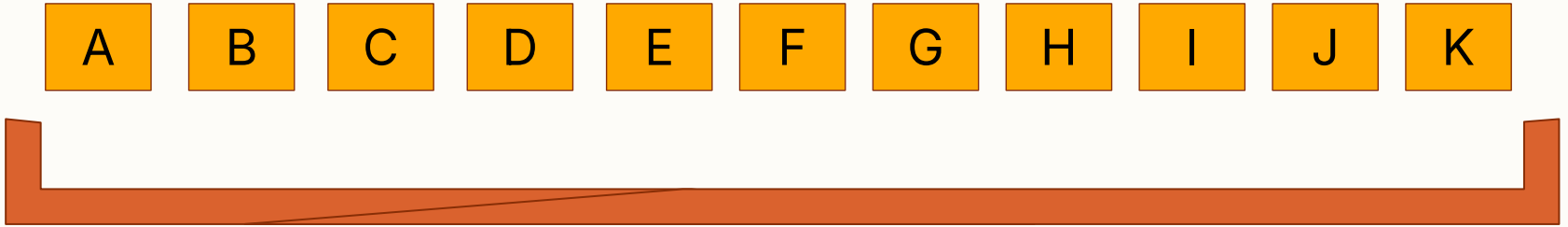
Searching

Target = J



Searching

Target = J



Left

Mid

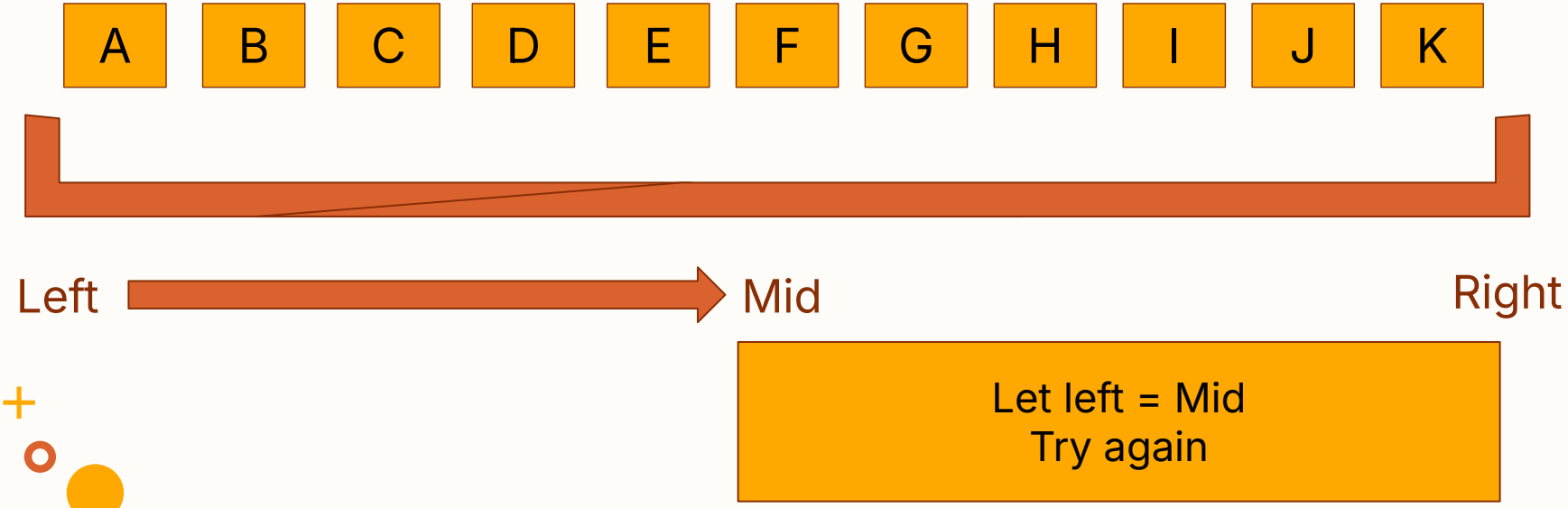
Right

“J” must be in this range

Since J > F

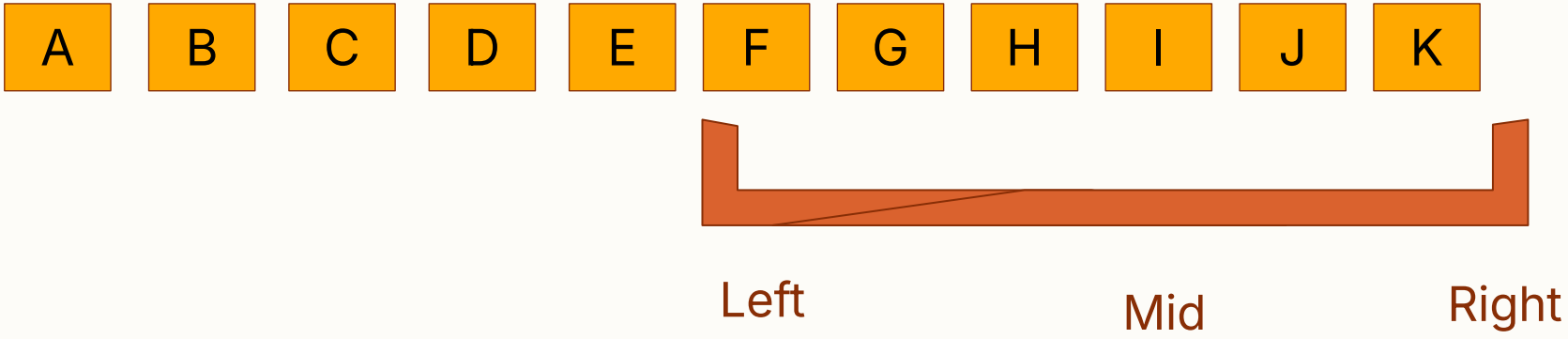
Searching

Target = J



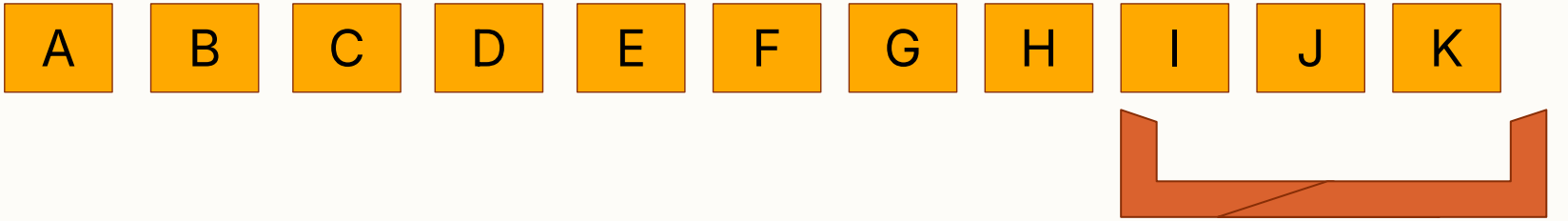
Searching

Target = J



Searching

Target = J



Eventually, Mid should equal the target, if it exists

Searching

- This type of search is called “Binary Search”
- Cut the space in half each time
  - Much more efficient
- Let’s code it
  - Count the operations
  - Inherently self similar

Searching

- Much faster than Linear Search
- If you have N elements, you need to do approximately log(N) operations
- O(logN)

Searching

- This comes up all the time
- Very commonly used
- Only possible is list is sorted

Afterwards I found a chatroom thread among Cambridge computer scientists, one of whom had also been told that unless he could pin down the moment of theft no one would look at the footage. He said he had tried to explain sorting algorithms to police — he was a computer scientist, after all.

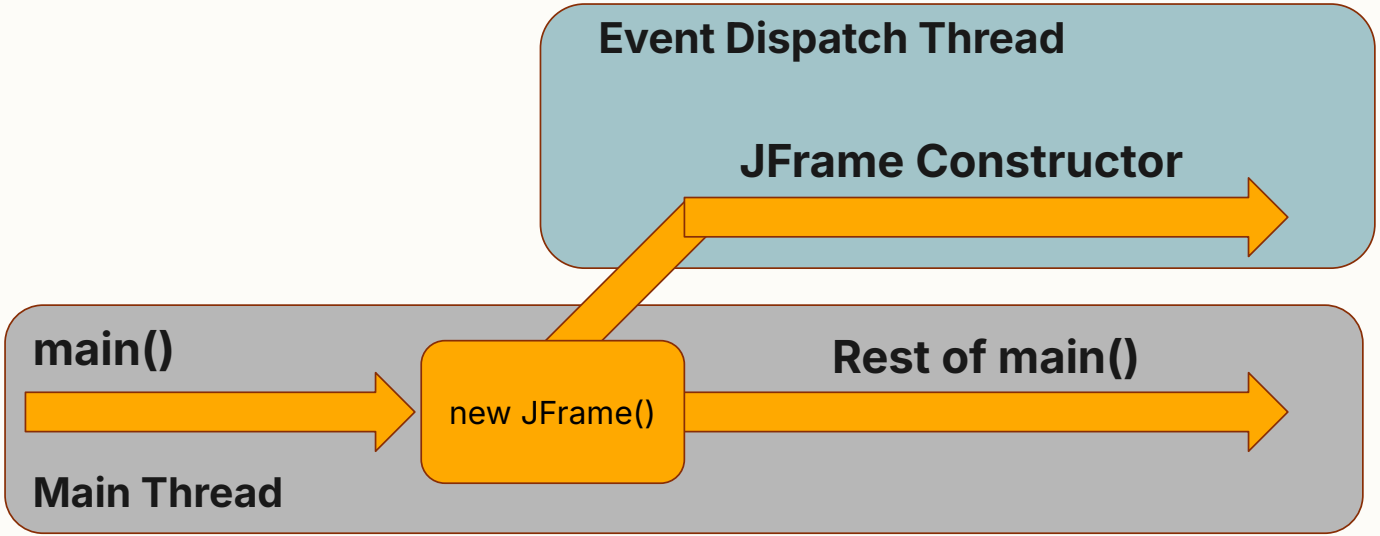
You don't watch the whole thing, he said. You use a binary search. You fast forward to halfway, see if the bike is there and, if it is, zoom to three quarters of the way through. But if it wasn't there at the halfway mark, you rewind to a quarter of the way through. It's very quick. In fact, he had pointed out, if the CCTV footage stretched back to the dawn of humanity it would probably have only taken an hour to find the moment of theft. This argument didn't go down well.

Threading

Reese Hatfield

Threading

- Remember the Event Dispatch Thread?
- Let your Swing code run in parallel



Threading

- Java lets you write arbitrary code that runs on a separate thread(s)

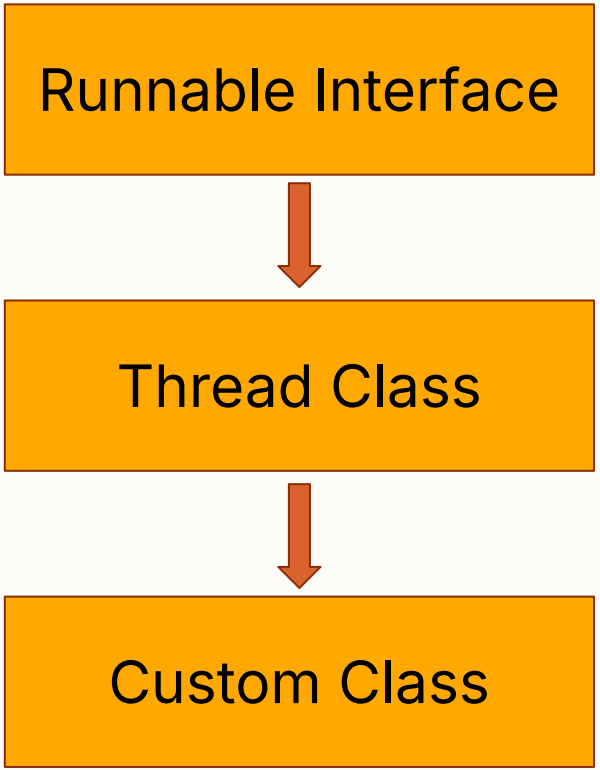
- Two primary ways of doing this
  - Thread Class
  - Runnable interface

Threading

- Anytime you want to write some code that will run asynchronously
  - Must at least *start* in a public void run() method

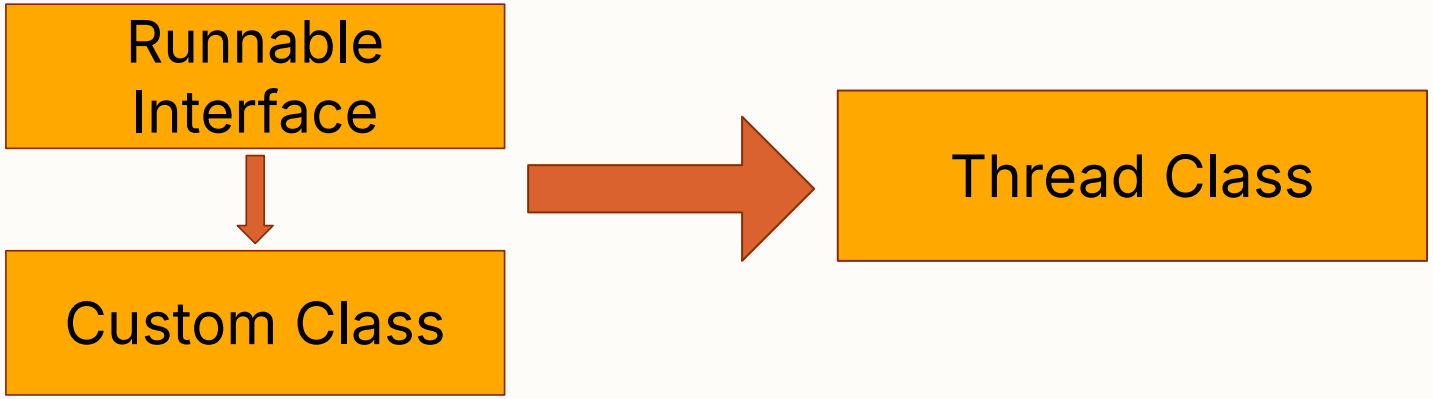
Threading

- Override the run() method from the Thread class
- Requires you to extend Thread



Threading

- You implement the runnable interface
- Pass that into a Thread instance



Threading

- Let’s try doing these
  - Gonna focus on extending from Thread
  - Likely what you’ll need for Project 4

Threading

- Threads are created like any other Object
  - Thread t1 = new CustomThread

- Do *NOT* call the run() method directly
  - Start threads via their .start() method
- Do *NOT* kill a thread directly
  - Wait for them to finish

Threading

- When you start() a thread:
  - The JVM will handle calling the run method in the background
  - Your code will continue to run after
  - It will *not* wait until it is finished

Threading

- When you join() a thread:
  - Your code will completely stop
  - Until that thread has finished all of its work
  - (block until run is finished)

Threading

- With just being able to
  - Create a thread
  - Start a thread
  - Join a thread

- You can do *basically* anything* in parallel

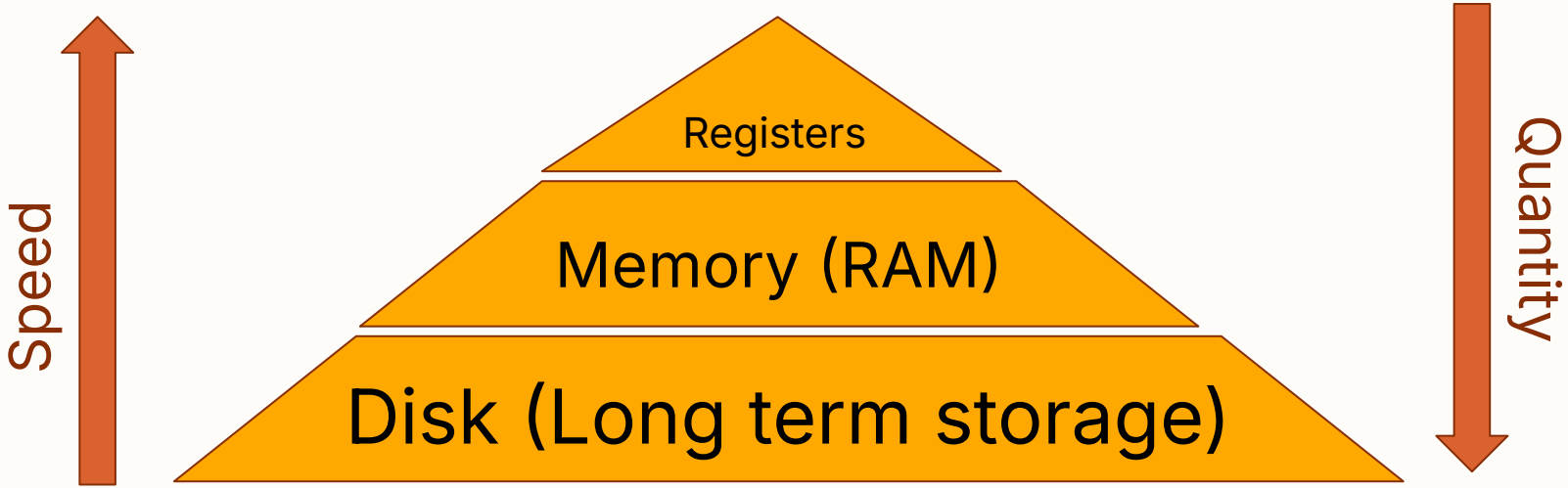
Threading

- How would threads work with the *static* keyword?

- Static = bound to class, not instance of class
- Effectively a single instance

Threading

- CPUs are *really* effective at quick data looku
  - This is one of their main purposes
- They will often cache data into registers



Threading

- Each thread can be thought of as a virtual CPU
- But if we want to share data across threads
  - A cache would cause our data be bad
  - Even if it’s static
- To fix this, we use the *volatile* keywords
  - Often alongside the *static* keyword

Threading

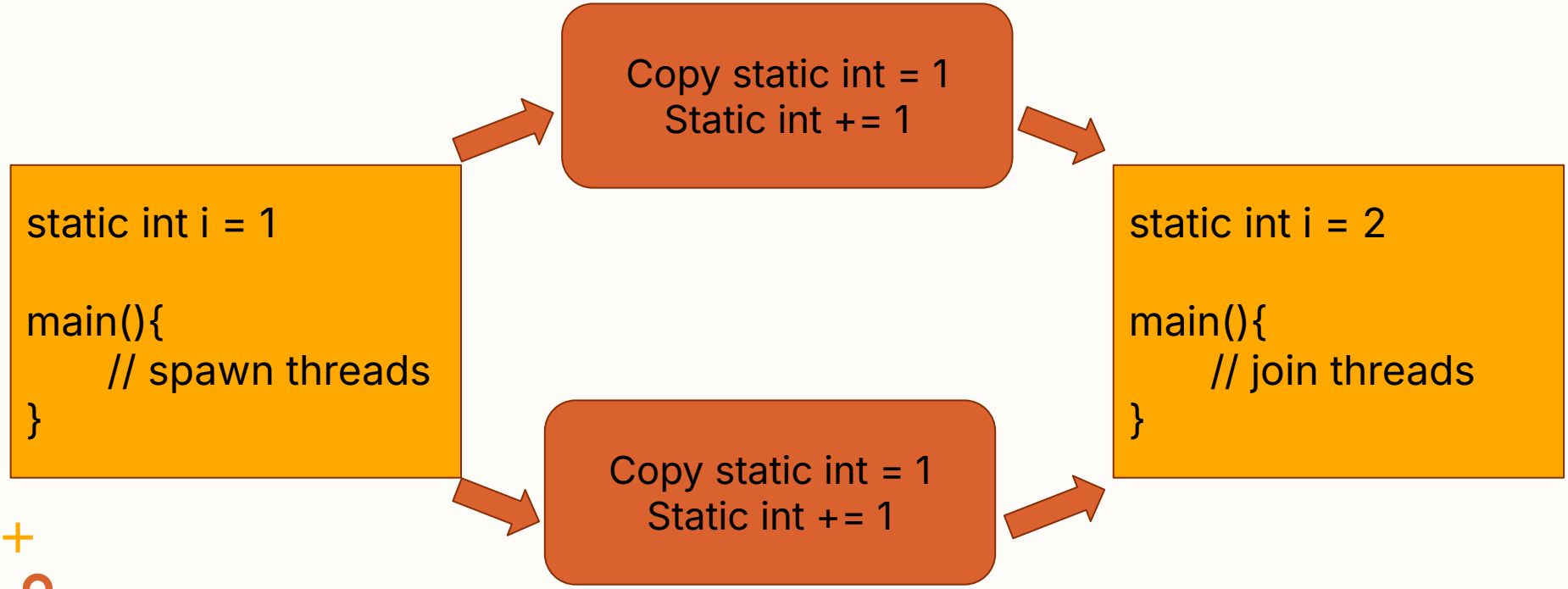
- We saw how to *use* volatile
  - And when to use it
  - But what happens if we *don't*
- Lets try it:
  - Spawn a thread → do some work for a tim
- After a second
  - Stop it from doing work in the main thread
  - Control work from a boolean

Threading

- Why does this break without *volatile*?
- A Thread gets its own environment
  - Separate variables, callstack, etc

- “Thread” will cache the value of “running”
- We need to tell it *not* to cache that value

Threading


static int i = 1
main(){
// spawn threads
}
Copy static int = 1
Static int += 1
Copy static int = 1
Static int += 1
static int i = 2
main(){
// join threads
}

Threading

- When should I use threading?
  - I/O bound operations
    - Files, etc
  - CPU bound operations
    - Must be “independent” operations

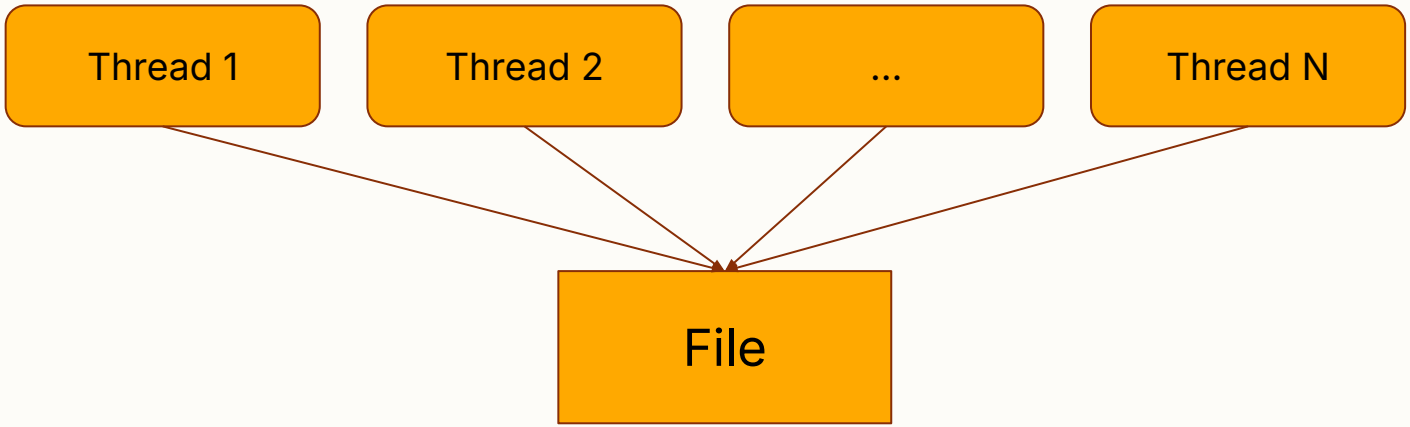
I/O Bound Threading

- I/O bound operations

- Reading/Writing to Files
  - Incredibly slow → see pyramid
  - Make multiple files
  - Have different threads access copies

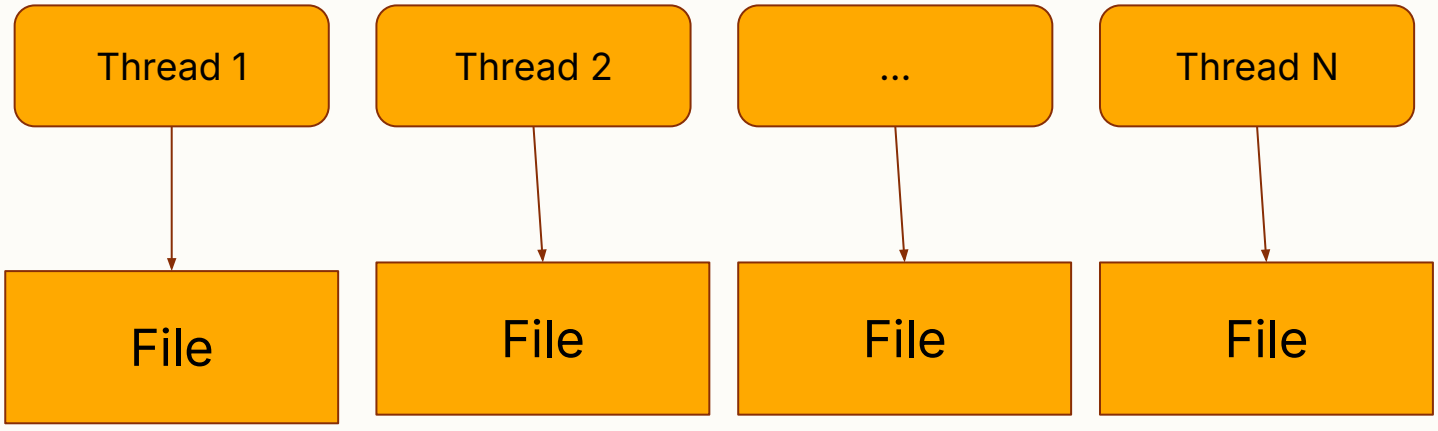
I/O Bound Threading

- Files can only handle one operation at a time
- Threads must wait on each other



I/O Bound Threading

- To fix this bottleneck:
  - Copy the files for each thread



CPU Bound Threading

- CPU are really good at executing sequential instructions
- Sequence = Ordered

- But we need thing to run in parallel*

CPU Bound Threading

- Most things cannot inherently be parallelized
- Most, not all

- How do we know if a task can be parallelized?

CPU Bound Threading

- Task can be done concurrently if:
- Task is sufficiently independent

- One piece does not depend on the rest
  - Taking a sum ⇒ A + B = B + A
  - Searching in a list

CPU Bound Threading

- Threading is also often used for long, blocking tasks
- If you were doing an operation, just once, but that operation took forever...
- Throw that task on another thread

Thread Concerns

- We saw before that
  - Need* a volatile variable to share data
  - Usually statically access
- This ensures each thread uses the same data
  - Why did we need this?

Thread Concerns

- Each thread would cache the variables value without volatile
- Is there ever a better way to do this?

- What if we wanted to sync our threads at the method level
  - Instead of variable access level

Thread Concerns

- There is a keyword for this
  - synchronized
- Prevent threads from interfering with inconsistent memory
- Let’s see this in action
  - Counter Class
  - Synchronized vs unsynchronized methods

Thread Safety

- By making our accessors synchronized
  - We made this class “thread safe”
  - Meaning, that our class state is always safe to access, even among different threads
- You’ll frequently see this written in documentation

Thread Safety

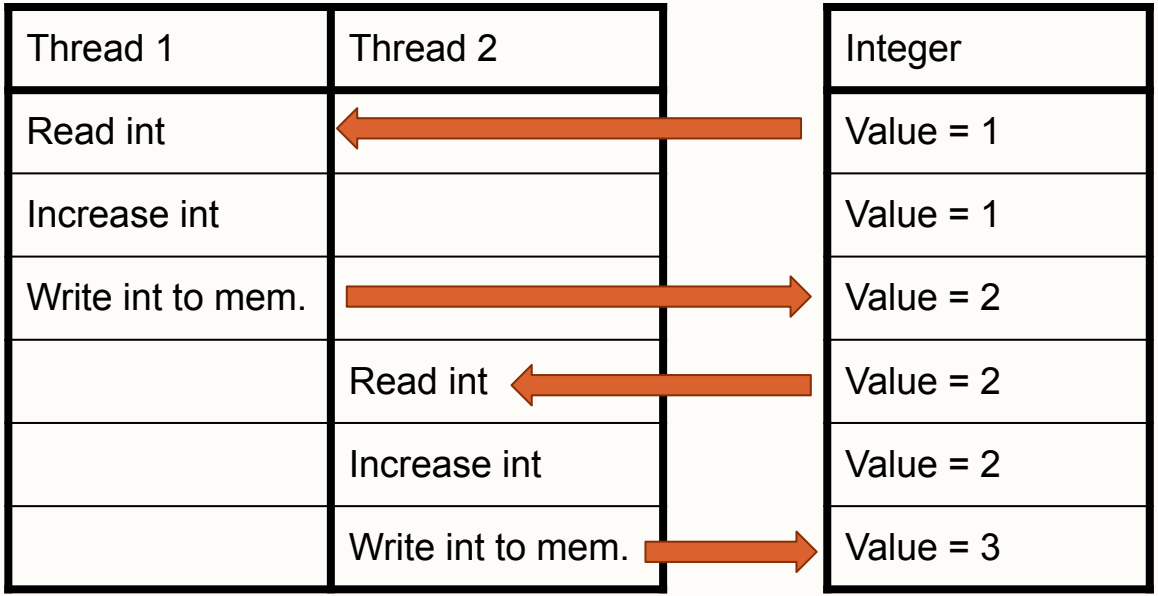
- Thread safety ensure our memory is safe to access

- If different threads did not promise this, whoever reached the data last, will have the wrong result

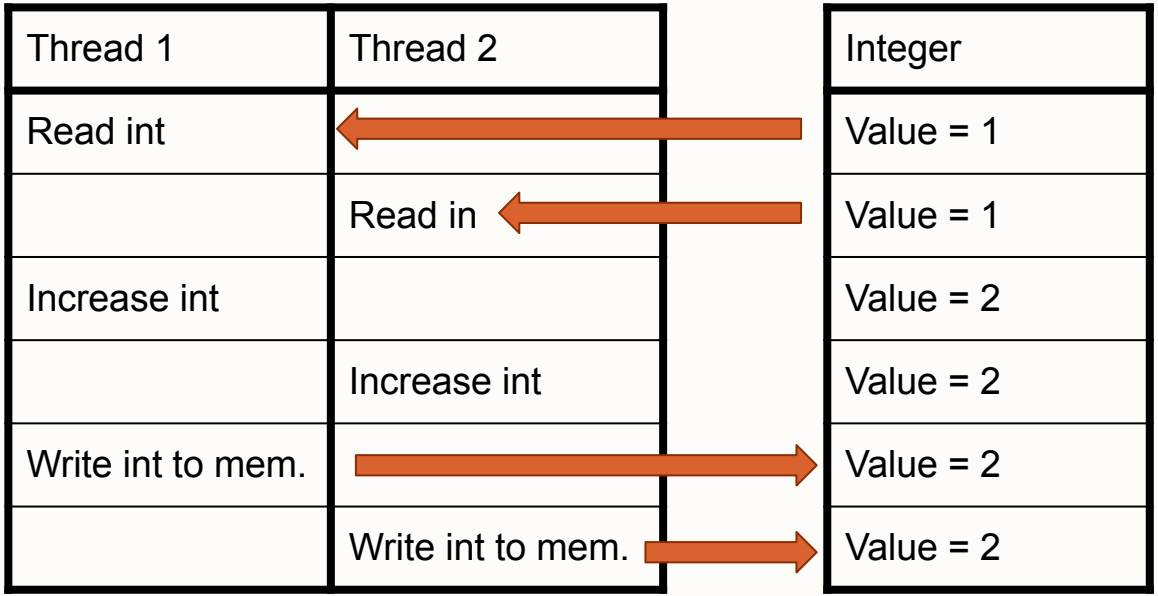
Thread Safety

- This is called a Race Condition
- Let’s visualize this a little better
- CPU’s execute sequential instructions
  - Read a value
  - Modify a value
  - Write to a value

Thread Safety



Thread Safety



Thread Safety

- The threads here ran at the same type
  - By sort of jumbling the timing of their instructions
  - This is a slightly more accurate depiction of what goes on
- Prevented by synchronized + volatile

Patterns

- The most common pattern you'll see in threading is "chunking"
  - Divide data into *variable* size chunks
  - Control for the number of threads

- Let's try it!