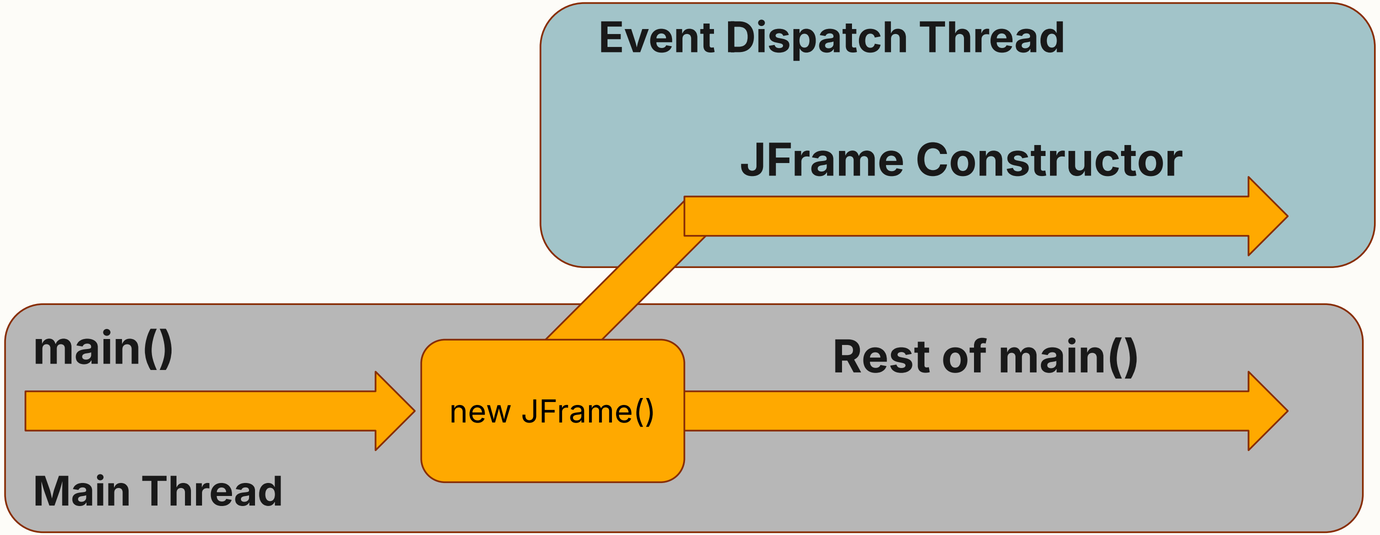
Threading

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

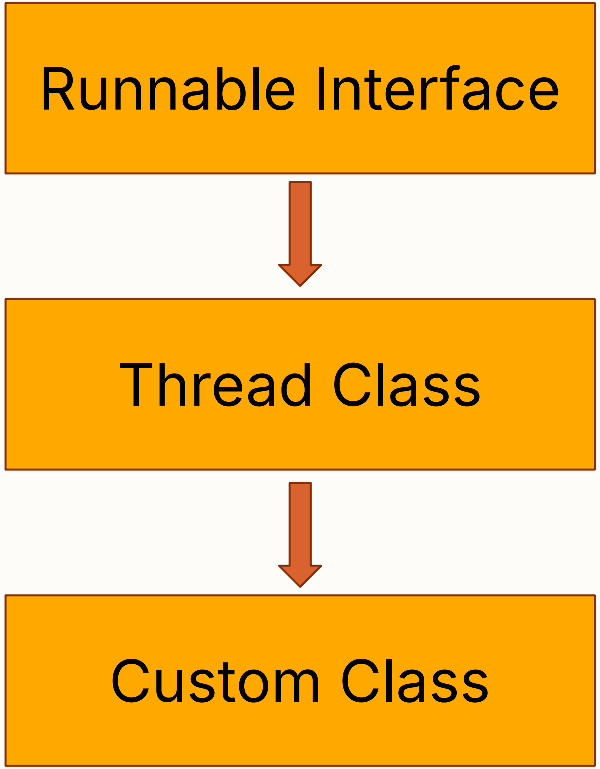
Threading

- 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



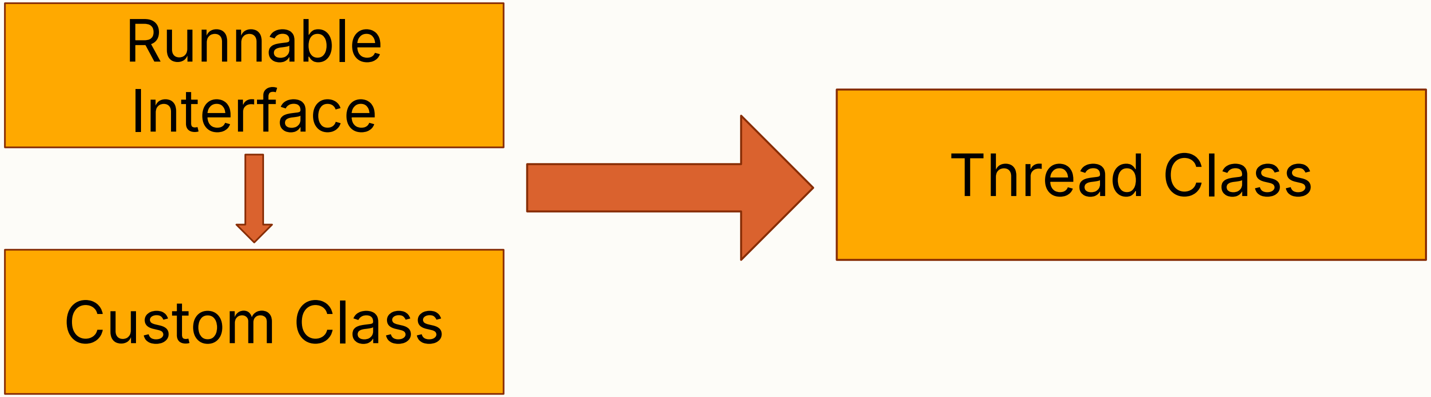
Runnable Interface

Thread Class

Custom Class

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

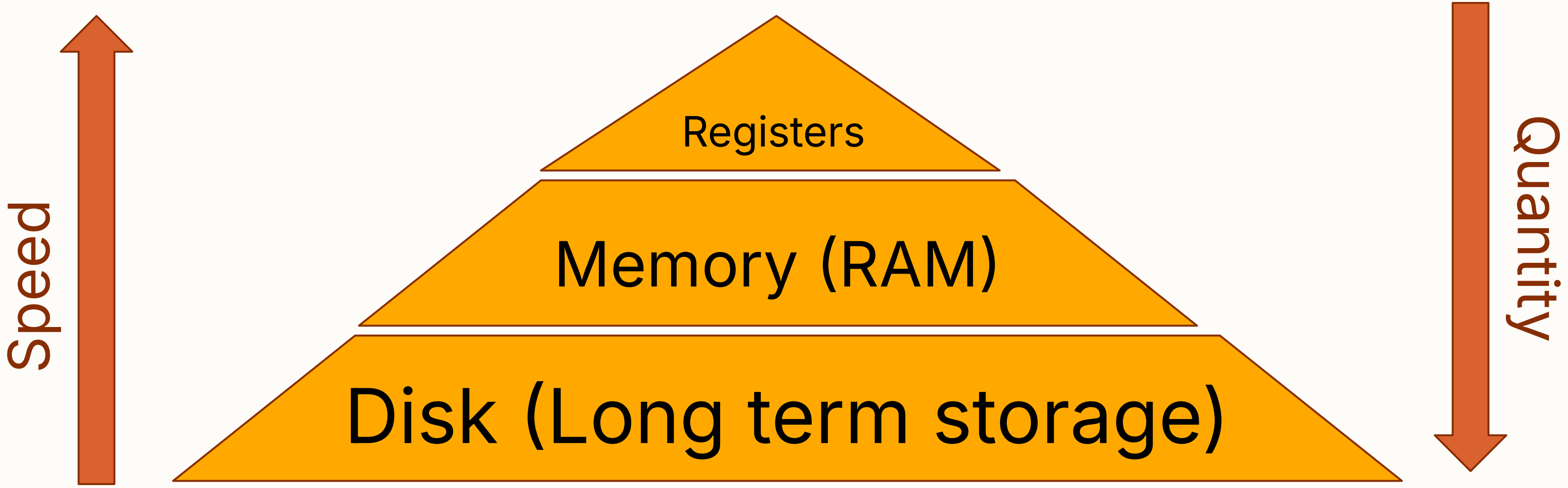
Threading

- 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



Speed

Registers

Memory (RAM)

Disk (Long term storage)

Quantity

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

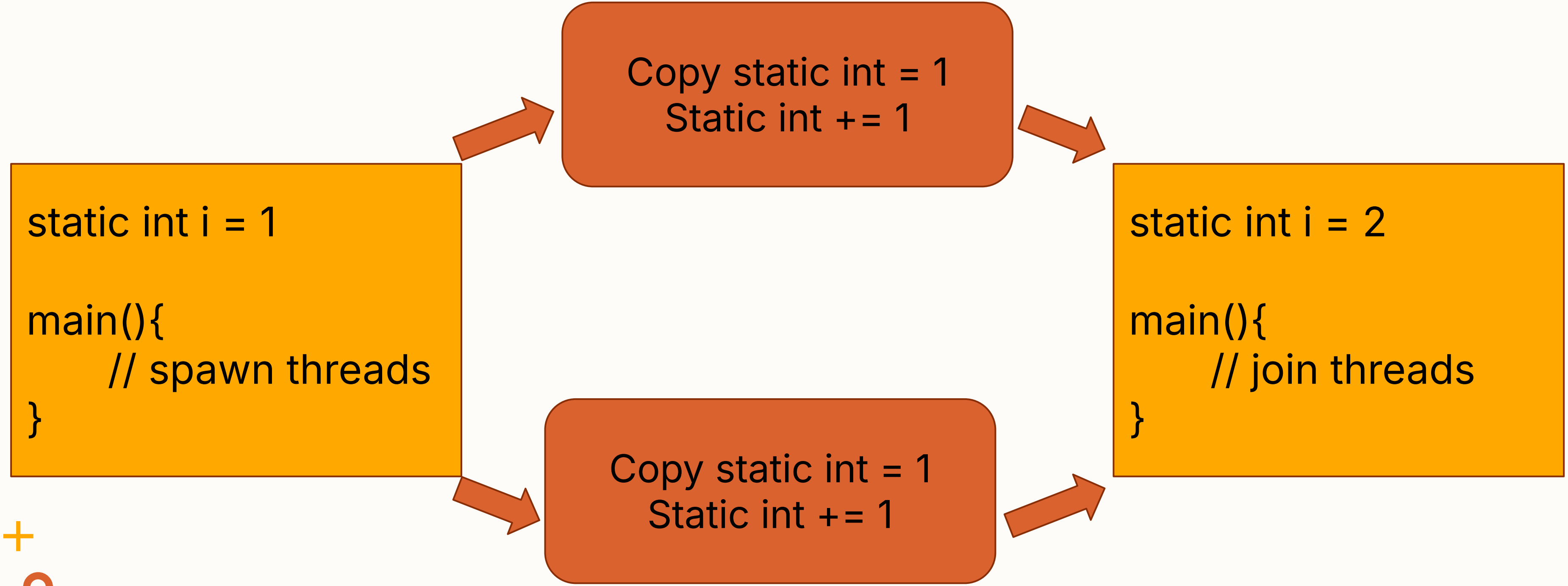
- We saw how to *use* volatile
  - And when to use it
  - But what happens if we *don't*
- Lets try it:
  - Spawn a thread → do some work for a tim
- After a second
  - Stop it from doing work in the main threa
  - Control work from a boolean

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

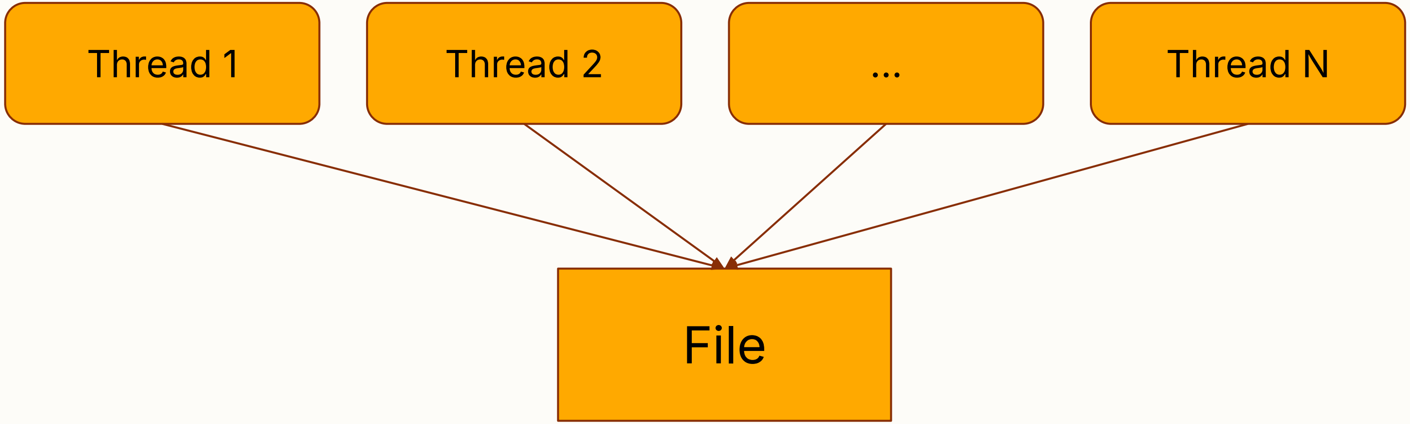
I/O Bound Threading

- I/O bound operations

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

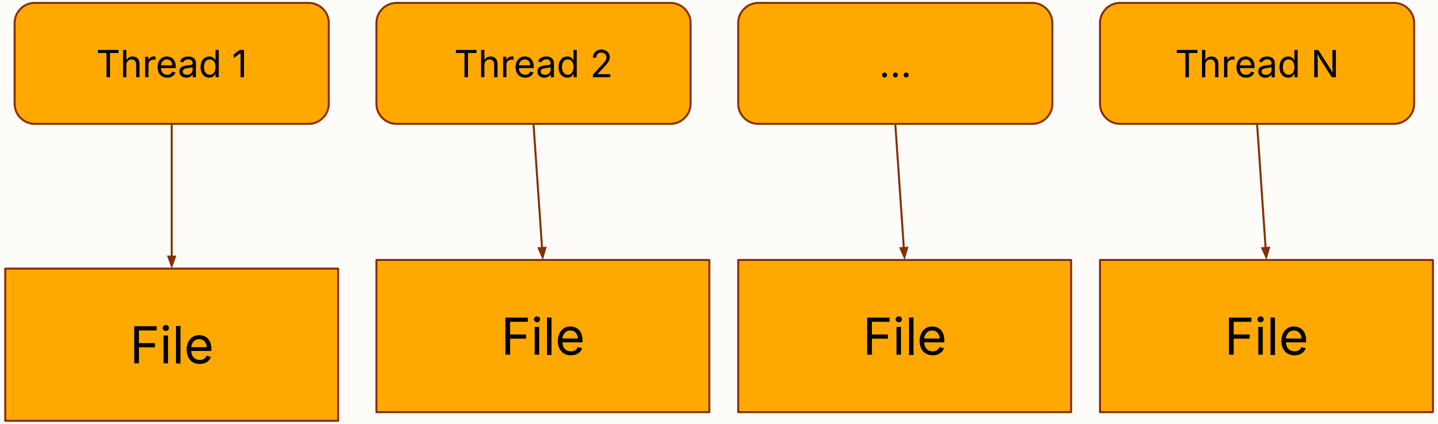
I/O Bound Threading

- 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

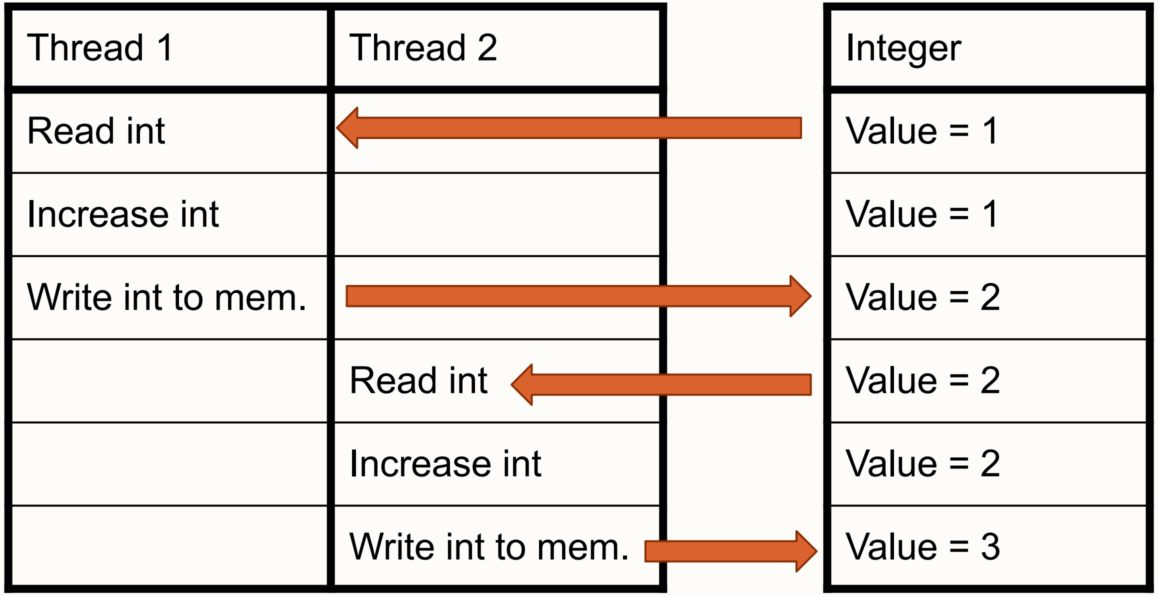
Thread Safety

- 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

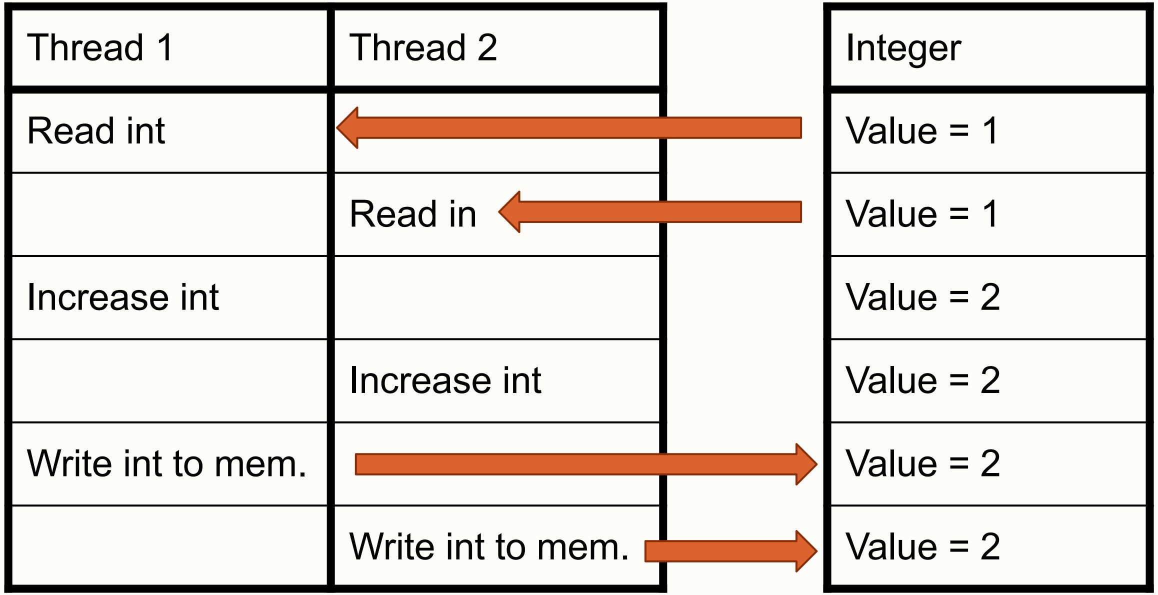
- This is called a Race Condition
- Let's visualize this a little better
- CPU's execute sequential instructions
  - Read a value
  - Modify a value
  - Write to a value

Thread Safety



Thread 1	Thread 2	Integer
Read int		Value = 1
Increase int		Value = 1
Write int to mem.		Value = 2
	Read int	Value = 2
	Increase int	Value = 2
	Write int to mem.	Value = 3

Thread Safety



Thread 1	Thread 2	Integer
Read int		Value = 1
	Read in	Value = 1
Increase int		Value = 2
	Increase int	Value = 2
Write int to mem.		Value = 2
	Write int to mem.	Value = 2

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!