Concurrency in Java

Prof. Stephen A. Edwards

The Java Language

- Developed by James Gosling et al. at Sun Microsystems in the early 1990s
- Originally called “Oak,” first intended application was as an OS for TV set top boxes
- Main goals were portability and safety
- Originally for embedded consumer software

The Java Language

- Set-top boxes: nobody cared
- Next big application: “applets”
  - Little programs dynamically added to web browsers
- Enormous Sun marketing blitz

- Partial failure:
  - Incompatible Java implementations
  - Few users had enough bandwidth
  - Fantastically slow Java interpreters
- Javascript has largely taken over this role
  - High-level scripting language
  - Has nothing to do with the Java language

The Java Language

- Where does Java succeed?

- Corporate programming
  - E.g., dynamic web page generation from large corporate databases in banks
  - Environment demands simpler language
    - Unskilled programmers, unreleased software
  - Speed, Space not critical
    - Tends to be run on very large servers
  - Main objective is reduced development time

The Java Language

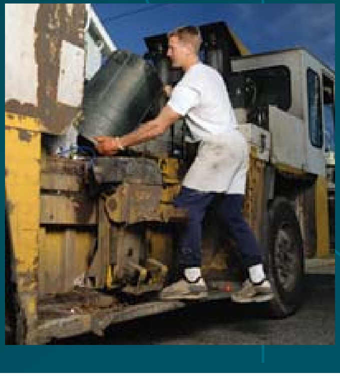
- Where does Java succeed?

- Education
  - Well-designed general-purpose programming language
  - Spares programmer from many common pitfalls
    - Uninitialized pointers
    - Memory management
  - Widely known and used, not just a teaching language

- Embedded Systems?
  - Jury is still out

Overview of Java

- Derived from C++, but incompatible
- Didn’t want to call it “C += 2”?
- No “loose” functions: everything part of a class
- Better package support (no preprocessor)
- Safer object references instead of pointers
- Large, powerful class library
- Automatic garbage collection
  - Programmer spared from memory management



Concurrency in Java

- Language supports threads
- Multiple contexts/program counters running within the same memory space
- All objects can be shared among threads
- Fundamentally nondeterministic
- Language provide some facilities to help avoid it

Thread Basics

- **How to create a thread:**

```
class MyThread extends Thread {
  public void run() { /* thread body */ }
}

MyThread mt = new MyThread;     // Create thread
mt.start();                // Starts thread running at run()
                           // Returns immediately
```

Thread Basics

- A thread is a separate program counter

  … and stack, local variables, etc.

- Not an object or a collection of things

- Classes, objects, methods, etc. do not belong to a thread

- Any method may be executed by one or more threads, even simultaneously

The Sleep Method

```
public void run() {
  for(;;) {
    try {
      sleep(1000); // Pause for 1 second
    } catch (InterruptedException e) {
       return;        // caused by thread.interrupt()
    }
    System.out.println(“Tick”);
  }
}
```

The Sleep Method

Does this print Tick once a second? No.

sleep() delay a lower bound

Rest of loop takes indeterminate amount of time

```
public void run() {
  for(;;) {
    try {
      sleep(1000);
    } catch (InterruptedException e) {
       return;
    }
    System.out.println(“Tick”);
  }
}
```

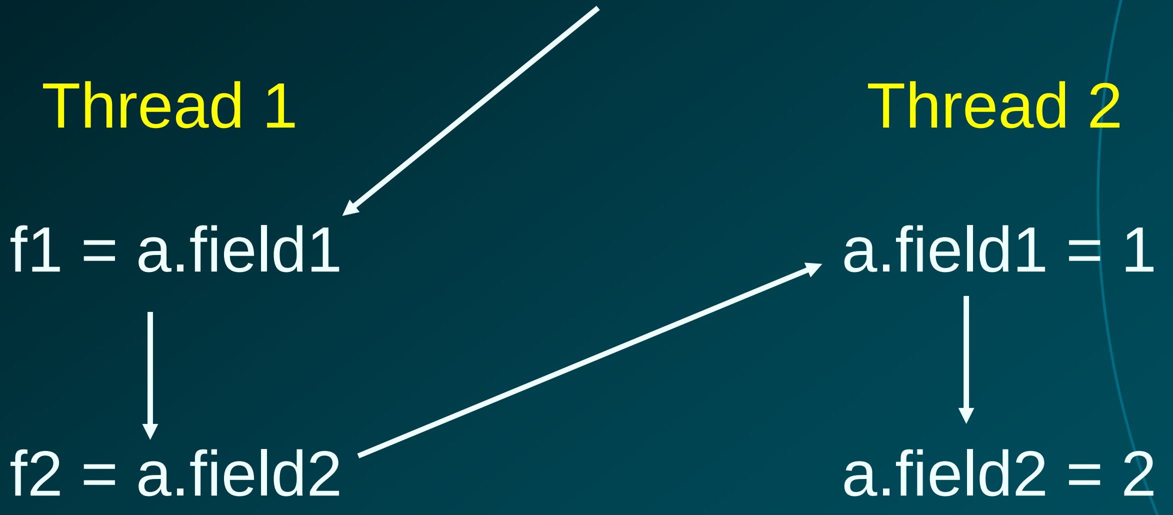
Races

- In a concurrent world, always assume someone else is accessing your objects
- Other threads are your adversary
- Consider what can happen when simultaneously reading and writing:

Thread 1	Thread 2
f1 = a.field1	a.field1 = 1
f2 = a.field2	a.field2 = 2

Races

- Thread 1 goes first
- Thread 1 reads original values



Thread 1

f1 = a.field1

f2 = a.field2

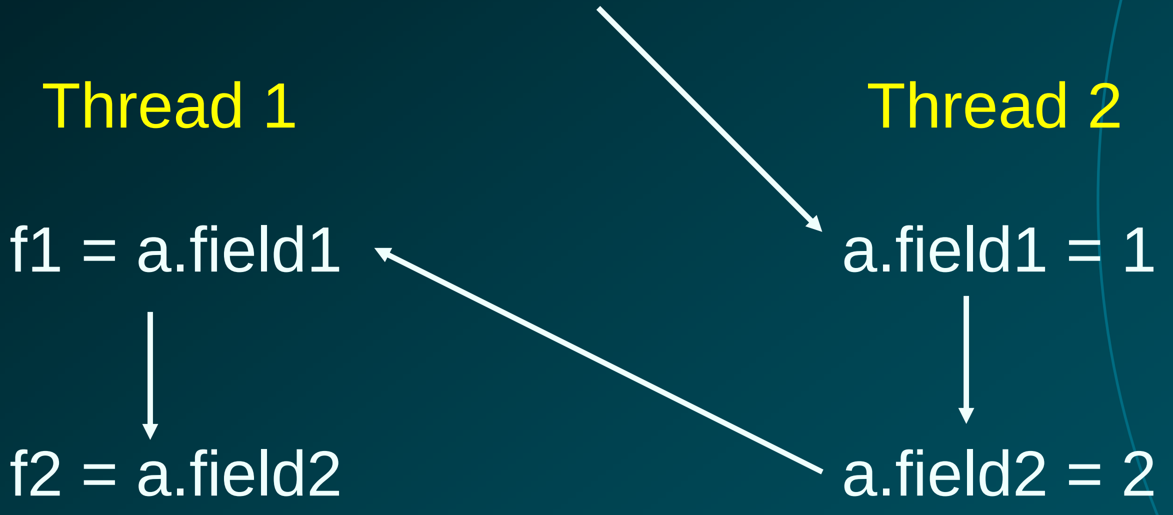
Thread 2

a.field1 = 1

a.field2 = 2

Races

- Thread 2 goes first
- Thread 1 reads new values



Thread 1

f1 = a.field1

f2 = a.field2

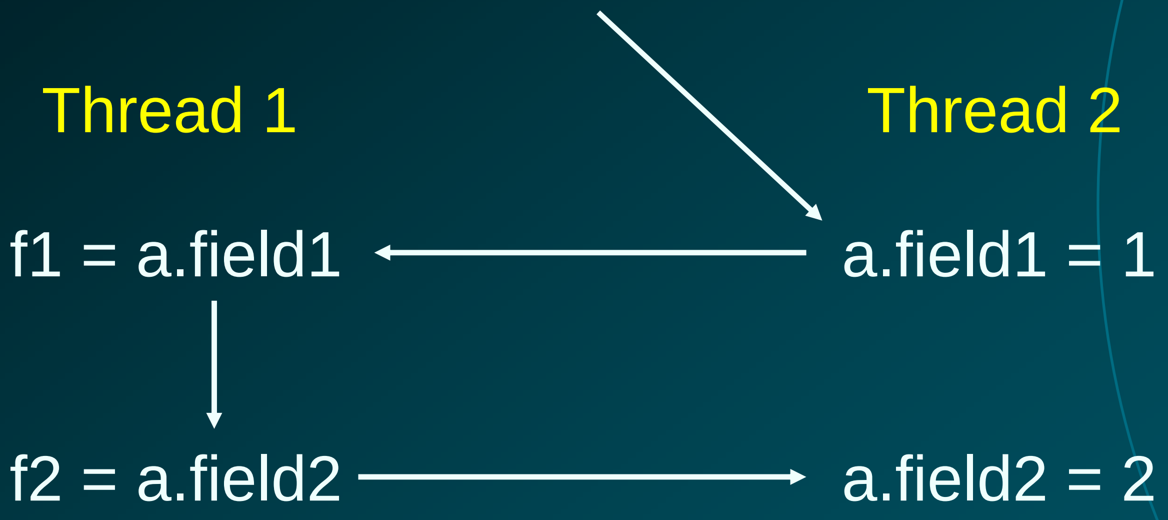
Thread 2

a.field1 = 1

a.field2 = 2

Races

- **Interleaved execution**
- **Thread 1 sees one new value, one old value**



Thread 1

Thread 2

f1 = a.field1

a.field1 = 1

f2 = a.field2

a.field2 = 2

Non-atomic Operations

- 32-bit reads and writes are guaranteed atomic
- 64-bit operations may not be

- Therefore,

int i; double d;

Thread 1	Thread 2	
i = 10;	i = 20;	i will contain 10 or 20
d = 10.0;	d = 20.0;	i might contain garbage

Per-Object Locks

- Each Java object has a lock that may be owned by at least one thread
- A thread waits if it attempts to obtain an already-obtained lock
- The lock is a counter: one thread may lock an object more than once

The Synchronized Statement

- A synchronized statement gets an object’s lock before running its body

```
Counter mycount = new Counter;
synchronized(mycount) {
 mycount.count();
}
```

“get the lock for mycount before calling count()”

- Releases the lock when the body terminates
- Choice of object to lock is by convention

Synchronized Methods

```
class AtomicCounter {
  private int _count;

  public synchronized void count() {
    _count++;
  }
}
```

“get the lock for the AtomicCounter object before running this method”

This implementation guarantees at most one thread can increment the counter at any time

Deadlock

```
synchronized(Foo) {
  synchronized(Bar) {
    /* Deadlocked */
  }
}
```

```
synchronized(Bar) {
  synchronized(Foo) {
    /* Deadlocked */
  }
}
```

- Rule: always acquire locks in the same order

Priorities

- Each thread has a priority from 1 to 10 (5 typical)
- Scheduler's job is to keep highest-priority threads running
- thread.setPriority(5)

What the Language Spec. Says

- From *The Java Language Specification*

> Every thread has a *priority*. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

- Vague enough for you?

Multiple threads at same priority?

- Language gives implementer freedom
- Calling yield() suspends current thread to allow other at same priority to run … maybe
- Solaris implementation runs threads until they stop themselves (wait(), yield(), etc.)
- Windows implementation timeslices

Starvation

- Not a fair scheduler
- Higher-priority threads can consume all resources, prevent lower-priority threads from running
- This is called starvation
- Timing dependent: function of program, hardware, and Java implementation

Waiting for a Condition

- Say you want a thread to wait for a condition before proceeding
- An infinite loop may deadlock the system

```
while (!condition) {}
```

- Yielding avoids deadlock, but is very inefficient

```
while (!condition) yield();
```

Java’s Solution: wait() and notify()

- wait() like yield(), but requires other thread to reawaken it

```
while (!condition) wait();
```

- Thread that might affect this condition calls() notify to resume the thread

- Programmer responsible for ensuring each wait() has a matching notify()

wait() and notify()

- Each object has a set of threads that are waiting for its lock (its wait set)

```
synchronized (obj) {          // Acquire lock on obj
  obj.wait();                 // suspend
                              // add thread to obj’s wait set
                              // relinquish locks on obj
```

In other thread:

```
obj.notify();                 // enable some waiting thread
```

wait() and notify()

1. Thread 1 acquires lock on object
2. Thread 1 calls wait() on object
3. Thread 1 releases lock on object, adds itself to object’s wait set
4. Thread 2 calls notify() on object (must own lock)
5. Thread 1 is reawakened: it was in object’s wait set
6. Thread 1 reacquires lock on object
7. Thread 1 continues from the wait()

wait() and notify()

- Confusing enough?
- notify() non-deterministically chooses one thread to reawaken (may be many waiting on same object)
  - What happens when there's more than one?
- notifyAll() enables all waiting threads
  - Much safer?

Building a Blocking Buffer

```
class OnePlace {
  El value;

  public synchronized void write(El e) { … }
  public synchronized El read() { … }
}
```

- Idea: One thread at a time can write to or read from the buffer

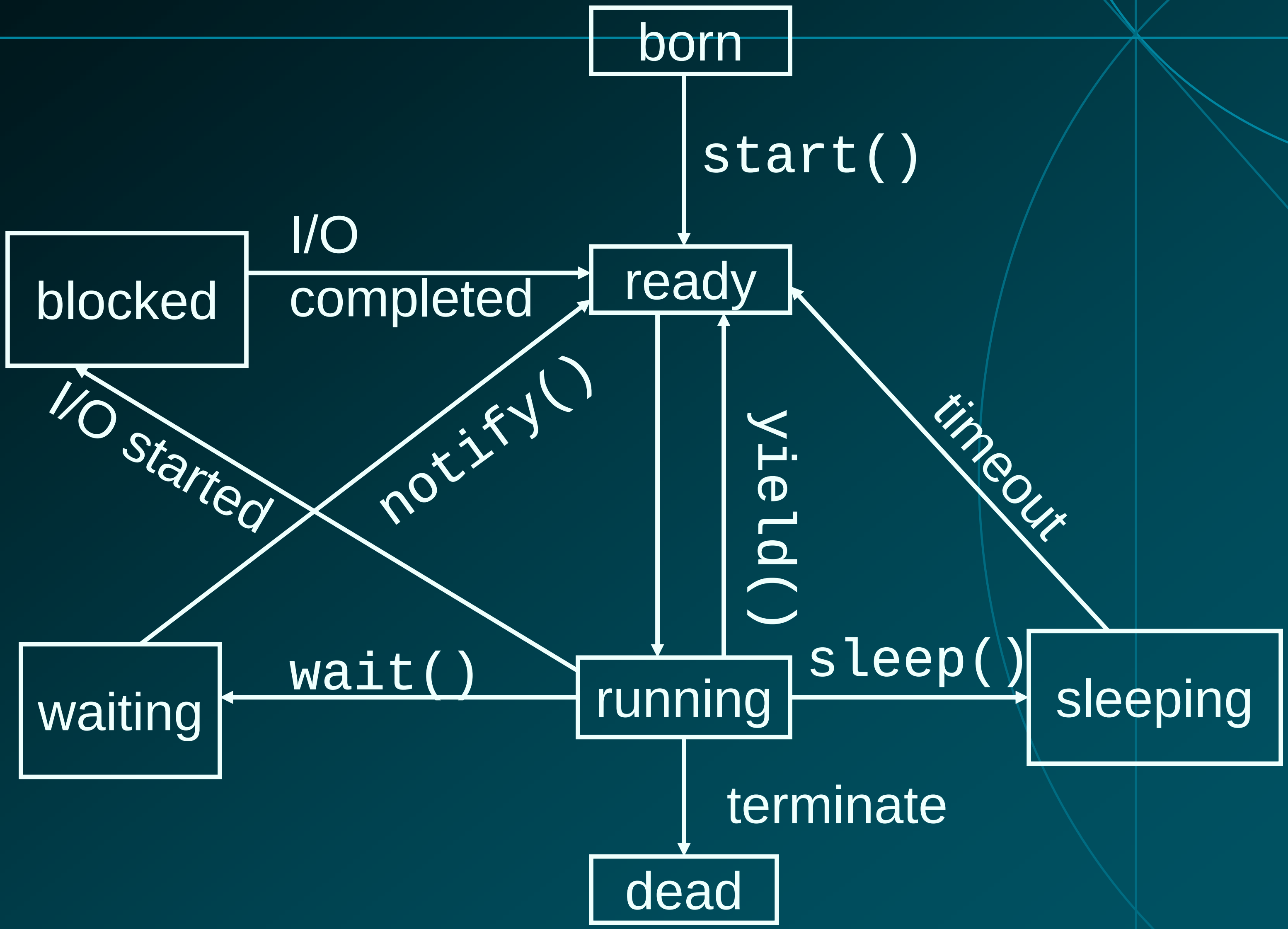
- Thread will block on read if no data is available
- Thread will block on write if data has not been read

Building a Blocking Buffer

```
synchronized void write(El e) throws InterruptedException
{
  while (value != null) wait();       // Block while full
  value = e;
  notifyAll();                        // Awaken any waiting read
}

public synchronized El read() throws InterruptedException
{
  while (value == null) wait();       // Block while empty
  El e = value; value = null;
  notifyAll();                        // Awaken any waiting write
  return e;
}
```

Thread States


born
start()
ready
blocked
I/O
completed
I/O started
notify()
yield()
timeout
waiting
wait()
running
sleep()
sleeping
terminate
dead