Synchronization Style

- Internal synchronization (class is thread-safe)
  - Have a stateful object synchronize itself (e.g., with synchronized methods). Robust to threaded callers.
  - E.g., class Random
- External synchronization (class is thread-compatible)
  - Have callers perform synchronization before calling the object. If they don’t, could be big problems.
  - Should be written to support atomicity.
  - E.g. ArrayList, HashMap

```
public class State {
    private int cnt = 0;
    public int synchronized incCnt(int x) {
        cnt += x;
    }
    public int synchronized getCnt() { return cnt; }
}
```

```
public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        s.incCnt(1);
    }
    public void main(String args[]) {
        State s = new State();
        MyThread thread1 = new MyThread(s);
        MyThread thread2 = new MyThread(s);
        thread1.start(); thread2.start();
    }
}
```

Synchronization occurs in State object itself, rather than in its caller.
public class MyThread extends Thread {
    static List l = new ArrayList();
    String s; // set in constructor
    void add(String s) {
        synchronized (l) { l.add(s); }
    }
    boolean check(String s) {
        synchronized (l) {
            return l.contains(s);
        }
    }
    public void run() {
        if (!check(s)) add(s);
    }
    public void main(String args[]) {
        MyThread thread1 = new MyThread("hello");
        MyThread thread2 = new MyThread("hello");
        MyThread thread3 = new MyThread("goodbye");
        thread1.start(); thread2.start();thread3.start();
    }
}

Synchronization occurs in the caller of ArrayList (which is MyThread), not ArrayList itself. But: there's a problem...

Lack of data races = atomicity?

• For the previous example:
  – Are there any data races?
  – What will value will the static variable l have at the end of an execution?
Answer

- There are no data races.
- But there is a violation of atomicity. We expect the output to be
  - { “hello”, “goodbye” }
- But in fact it could also be
  - { “hello”, “hello”, “goodbye” }
- Fix:
  - The check() and add() methods must be called indivisibly

Thread-Compatible class fixed

```java
public class MyThread extends Thread {
    static List l = new ArrayList();
    String s;
    public void run() {
        synchronized (l) {
            if (!l.contains(s))
                l.add(s);
        }
    }

    public void main(String args[]) {
        MyThread thread1 = new MyThread("hello");
        MyThread thread2 = new MyThread("hello");
        MyThread thread3 = new MyThread("goodbye");
        thread1.start(); thread2.start();
        thread3.start();
    }
}
```

Both contains() and add() are now guarded by a single synchronized block, making them atomic
String class

- Is the String class thread-safe or thread-compatible?
  - Fact: none of its methods are annotated with the keyword “synchronized”

- Remember: the key difficulty with threads is mutation of shared state. Immutable shared state can never violate atomicity.
  - This is quite desirable, particularly since (next slide please …)

Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called deadlock

```java
Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}

T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
```
Deadlock

• Quite possible to create code that deadlocks
  – Thread 1 holds lock on A
  – Thread 2 holds lock on B
  – Thread 1 is trying to acquire a lock on B
  – Thread 2 is trying to acquire a lock on A
  – Deadlock!

• Not easy to detect when deadlock has occurred
  – Other than by the fact that nothing is happening

Deadlock: Wait graphs

- Thread T1 holds lock A
- Thread T2 attempting to acquire lock B

Deadlock occurs when there is a cycle in the graph
Wait graph example

T1 holds lock on A
T2 holds lock on B
T1 is trying to acquire a lock on B
T2 is trying to acquire a lock on A

Key Ideas

• Multiple threads can run simultaneously
  – Either truly in parallel on a multiprocessor
  – Or can be scheduled on a single processor
    • Assume a running thread can be preempted at any time

• Threads can share data
  – In Java, only fields can be shared
  – Need to prevent interference
    • Synchronization, immutability, and other methods
  – Overuse of synchronization can create deadlock
    • Violation of liveness
Guaranteeing Safety

- Ensure objects are accessible only when in a **consistent** and appropriate state
  - All invariants are maintained
  - Presents subclass obligations
- Use locks to enforce this
  - Rule of thumb 1: You must hold a lock when accessing shared data
  - Rule of thumb 2: You must not release a lock until shared data is in a valid state

Guaranteeing Liveness

- Ensuring availability of services
  - Called methods eventually execute
- Ensuring progress of activities
  - Managing resource contention
  - Freedom from deadlock
  - Fairness
  - Fault tolerance