public class Example extends Thread {
    private static int cnt = 0;  // shared state
    public void run() {
        int y = cnt;
        cnt = y + 1;
    }
    public static void main(String args[]) {
        Thread t1 = new Example();
        Thread t2 = new Example();
        t1.start();
        t2.start();
    }
}
Data Race Example

```
static int cnt = 0;  

t1.run() {
    int y = cnt;
    cnt = y + 1;
}

t2.run() {
    int y = cnt;
    cnt = y + 1;
}

Start: both threads ready to run. Each will increment the global count.
```
Data Race Example

static int cnt = 0;  

\[\text{T1 executes, grabbing the global counter value into } y.\]
Data Race Example

```
static int cnt = 0;  // Shared state cnt = 0

void t1.run() {
  int y = cnt;     // y = 0
  cnt = y + 1;
}

void t2.run() {
  int y = cnt;     // y = 0
  cnt = y + 1;
}
```

T1 is pre-empted. T2 executes, grabbing the global counter value into y.
Data Race Example

static int cnt = 0;  
Shared state  
\[ \text{cnt} = 1 \]

\[ \text{t1.run()} \{ \]
\[ \text{int y = cnt; } \quad \text{y = 0} \]
\[ \text{cnt = y + 1; } \]
\[ \} \]

\[ \text{t2.run()} \{ \]
\[ \text{int y = cnt; } \quad \text{y = 0} \]
\[ \text{cnt = y + 1; } \]
\[ \} \]

T2 executes, storing the incremented cnt value.
Data Race Example

```
static int cnt = 0;  // Shared state  cnt = 1

  t1.run() {  
    int y = cnt;  // y = 0
    cnt = y + 1;
  }

  t2.run() {  
    int y = cnt;  // y = 0
    cnt = y + 1;
  }

T2 completes.  T1 executes again, storing the old counter value (1) rather than the new one (2)!
```
But When I Run it Again?
Data Race Example

```
static int cnt = 0;

t1.run() {
    int y = cnt;
    cnt = y + 1;
}

t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

*Shared state*  \( cnt = 0 \)

*Start: both threads ready to run. Each will increment the global count.*
Data Race Example

static int cnt = 0;  

Shared state  cnt = 0

T1 executes, grabbing the global counter value into y.
**Data Race Example**

```java
static int cnt = 0;  // Shared state  cnt = 1

t1.run() {
    int y = cnt;   // y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;   // y = 0
    cnt = y + 1;
}
```

- T1 executes again, storing the counter value

```
46
```
Data Race Example

```java
static int cnt = 0;  // Shared state  cnt = 1

t1.run() {
    int y = cnt;    // y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;    // y = 1
    cnt = y + 1;
}
```

*T1 finishes.  T2 executes, grabbing the global counter value into y.*
Data Race Example

static int cnt = 0;           // Shared state  cnt = 2

t1.run() {
    int y = cnt;  y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;  y = 1
    cnt = y + 1;
}

T2 executes, storing the incremented cnt value.
What Happened?

• In the first example, \( t1 \) was preempted after it read the counter but before it stored the new value.
  – Depends on the idea of an \textit{atomic action}
  – Violated an object invariant

• A particular way in which the execution of two threads is interleaved is called a \textit{schedule}. We want to prevent this undesirable schedule.

• Undesirable schedules can be hard to reproduce, and so hard to debug.
Question

• If instead of
  
  ```
  int y = cnt;
  cnt = y+1;
  ```

• We had written
  
  ```
  cnt++;
  ```

• Would the result be any different?

• Answer:  NO!
  
  ```
  Don’t depend on your intuition about atomicity
  ```
If you run a program with a race condition, will you always get an unexpected result?

- No! It depends on the scheduler
- ...i.e., which JVM you’re running
- ...and on the other threads/processes/etc that are running on the same CPU

Race conditions are hard to find
What can this example generate?

```java
static int cnt = 2;
t1.run() {
    int y = cnt;
    cnt = y * 2;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```
public class Example extends Thread {
    private static int cnt = 0;
    static Object lock = new Object();
    public void run() {
        synchronized (lock) {
            int y = cnt;
            cnt = y + 1;
        }
    }
}

Lock, for protecting the shared state
Acquires the lock;
Only succeeds if not held by another thread
Releases the lock
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}

t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

*Shared state*  \( \text{cnt} = 0 \)

*T1 acquires the lock*
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;    y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
Applying Synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

**Shared state**

cnt = 0

**T1 is pre-empted.**

**T2 attempts to acquire the lock but fails because it’s held by T1, so it blocks**
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

*Shared state* \( \text{cnt} = 1 \)

*\( T1 \) runs, assigning to \( \text{cnt} \)*
int cnt = 0;

```
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  // y = 0
    }
}
```

t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}

*Shared state*  
\( cnt = 1 \)

*T1 releases the lock and terminates*
Applying Synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  // T2 now can acquire the lock.
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

**Shared state**  
\[cnt = 1\]
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1; y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1; y = 1
    }
}
```

Shared state  cnt = 1

T2 reads cnt into y.
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1; \(\text{y} = 0\)
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1; \(\text{y} = 1\)
    }
}

\textit{Shared state} \quad \texttt{cnt} = 2

\textit{T2 assigns} \texttt{cnt},
\textit{then releases the lock}
Locks

- *Any* Object subclass has (can act as) a lock
- Only one thread can hold the lock on an object
  - Other threads block until they can acquire it
- If a thread already holds the lock on an object
  - The thread can reacquire the same lock many times
    - ...Locks are *reentrant*
  - Lock is released when object unlocked the corresponding number of times
- No way to attempt to acquire a lock in Java 1.4
  - Either succeeds, or blocks the thread
Synchronized Statement

• `synchronized (obj) { statements }`
• Obtains the lock on `obj` before executing statements in block
• Releases the lock when the statement block completes
  – Either normally, or due to a return, break, or exception being thrown in the block
Synchronized Methods

- A method can be synchronized
  - Add `synchronized` modifier before return type
- Obtains the lock on object referenced by `this` before executing method
  - Releases lock when method completes
- For a `static synchronized` method
  - Locks the `Class` object for the class
    - Accessible directly, e.g. `Foo.class`
  - Not the same as `this` (there is no this)
Synchronization Example

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

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.
• The object locked, and the fields protected by the lock, may be different

• It is common to protect fields by holding a lock on the object they are fields of
  – but this is not required

• For example, fields of all the nodes of a tree might be protected by a lock on the tree
Synchronization Style

• Design decision
  – Internal synchronization (class is thread-safe)
    • Have a stateful object synchronize itself (e.g., with synchronized methods)
  – External synchronization (class is thread-compatible)
    • Have callers perform synchronization before calling the object

• Can go both ways:
  – Thread-safe: Random
  – Thread-compatible: ArrayList, HashMap, …
**Need for Synchronization**

- Need to use synchronization to ensure that two threads cannot access a shared memory location at the same time
  - Unless both are reading the memory
  - *volatile* is a special case we’ll come to later
What can go wrong?

- Deadlock
- Insufficient atomicity
- Non-determinism
- Just plain wrong
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
• Consider:
  – each bank account is protected by a distinct lock
  – Operations on an account acquire the lock on that account
  – You want to move money from one account to the other
    • You need to hold both locks
Avoiding deadlock

• Can try to never hold a lock on more than one object at a time
  – Be careful what you call while holding a lock

• Can impose a lock ordering
  – A total order such that if $X < Y$ and I hold a lock on both $X$ and $Y$, I grab the lock on $X$ before the lock on $Y$
public class State {
    private int cnt = 0;
    public synchronized int getCnt() {
        return cnt;
    }
    public synchronized void setCnt(int newValue) {
        cnt = newValue;
    }
}

public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        s.setCnt(s.getCnt()+1);
    }
    public void main(String args[]) {
        State s = new State();
        MyThread thread1 = new MyThread(s);
        MyThread thread2 = new MyThread(s);
        thread1.start(); thread2.start();
    }
}
Insufficient Atomicity

• In a number of situations, you will want to
  – Examine the current state of the system
  – Update the system based on that examination

• You write this assuming that this will be done as an atomic action
  – But if the operations to examine the system and update the system are separate synchronized methods, doesn’t work
public class WordCounter extends Thread {

    static Map<String, Integer> wordCount
        = Collections.synchronizedMap(
            new HashMap<String, Integer>());

    public void run() {
        while (...) {
            String word = ...;
            Integer v = wordCount.get(word);
            if (v == null) wordCount.put(word, 1);
            else wordCount.put(word, v+1);
        }
    }
}
Nondeterminism

- Even if your program is entirely "correct", it may be non-deterministic
  - Not consistently producing the same result

- Makes testing your code difficult
- Makes finding bugs difficult
- Generally, no fix for nondeterminism in multithreaded code
Just plain wrong

- Multithreaded code can be subtle, and testing is difficult
- Easy to make mistakes when writing multithreaded code
- Use building blocks built by experts if possible
  - java.util.concurrent