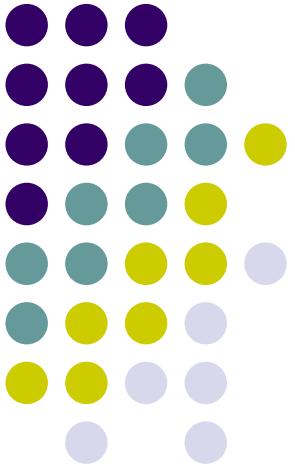
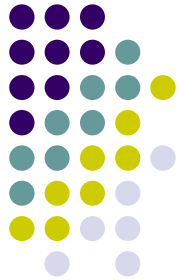
Java: Pitfalls and Strategies

Pao-Ann Hsiung
National Chung Cheng University
Chiayi, Taiwan



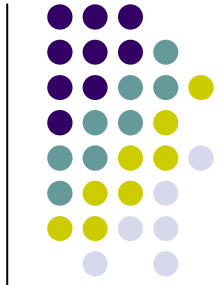
Adapted from Bo Sandén, “Copying with Java Threads,”
IEEE Computer, Vol. 37, No. 4, pp. 20-27, April 2004.

Contents



1. Multiple threads, one object
2. Omitting the synchronized keyword
3. Maintaining synchronized blocks
4. Wait loop placement
5. Missing notification of waiting threads
6. Confusing long and short waits

Pitfall #1:
Multiple threads, one object

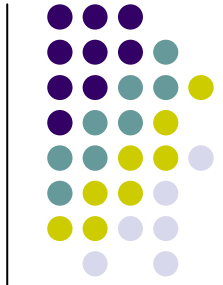


- Given a class R that implements **Runnable**
- 2 ways to create multiple threads executing R's run method:
  - Instantiate R n times, submit each instance once as a parameter to a Thread constructor
    - Own set of instance variables
  - Submit one R n times to Thread's constructor
    - Only one object → **inconsistencies** in object data

Strategy #1:
Multiple threads, one object

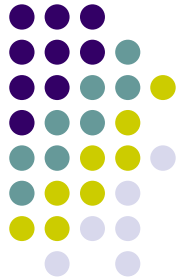
- Policy
  - Submit each runnable object to a Thread constructor only ONCE!
- Compliance checking
  - Include thread creation and start in the constructor
  - **new Thread(this).start()**

Pitfall #2:
Omitting the synchronized keyword



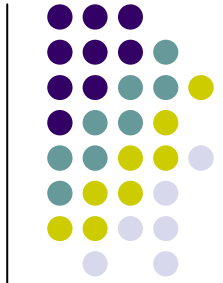
- Freedom to synchronize selected methods of a class
  - forgotten or purposely unsynchronized methods → data inconsistencies
  - Eg: get() synchronized, put() not synchronized
- Synchronized methods using public variables
  - another method changing the variables without obtaining object lock

Strategy #2:
Omitting the synchronized keyword



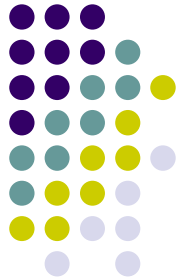
- A tool or compiler can identify and flag unsynchronized methods
- Difficult to spot methods that need to be synchronized

Pitfall #3:
Maintaining synchronized blocks



- Synchronized block
  - a synchronized method declared outside a class
  - may interpret a class as nonsynchronized (whereas it has methods synchronized in some block)
  - may add a method to a class without realizing it needs to be synchronized (due to the block synchronization)

Strategy #3:
Maintaining synchronized blocks

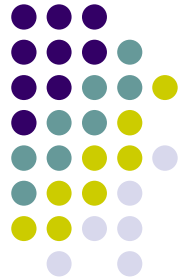


- Prefer synchronized methods to synchronized blocks
- Exception
  - Coordinating operations on different objects
  - Add comment to a class that has objects synchronized in blocks

Pitfall #4:
Wait loop placement (1/2)

- Wait loop should be written as:
  - while (cond) wait();
  - object retests cond after reactivation from wait set
- Variations:
  - while (cond) yield(); // obj lock not released
  - if (cond) wait(); // no retest after reactivation

Pitfall #4:
Wait loop placement (2/2)



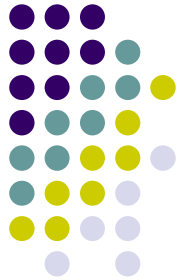
- Java allows wait loop to be placed anywhere in a critical section (synchronized method)
  - synchronized void m() {
  - callCounter++;
  - while (cond) wait();
  - if (callCounter==5)
  - …
  - }

Data obtained or changed **BEFORE** wait()

Data obtained or changed **AFTER** wait()

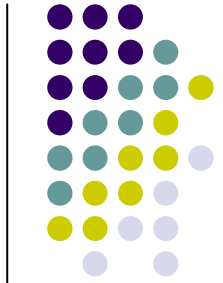
Data changed by some other method?

Strategy #4:
Wait loop placement



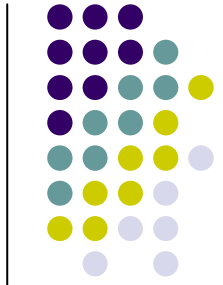
- Tool or compiler flags
  - incorrect wait()
  - any wait loop that is not the first statement in a synchronized method

Pitfall #5:
Missing notification of waiting threads



- Synchronizations
  - exclusion: automatic (synchronized)
  - condition: not automatic (wait, notify)
- notify
  - **standard Java**: arbitrary wait set
  - **real-time Java**: FIFO wait set
- notifyAll
  - useful when multiple threads can proceed
  - in standard Java, only way to activate highest priority thread
  - in standard and real-time Java, needed when threads pend on different conditions, to be sure of activating one pending on that condition

Strategy #5:
Missing notification of waiting threads

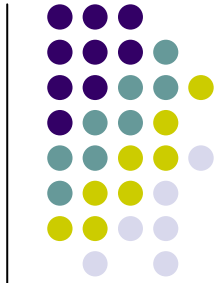


- Code inspection
  - check if all necessary notify, notifyAll have been called
- Time-out parameter in each wait()
  - thread automatically reactivated after time-out
  - checks condition and then
    - proceeds or
    - reenters wait set
  - Does not work for real-time Java!

Pitfall #6:
Confusing long and short waits (1/5)

- Waits:
  - **long wait** ➔ use condition synchronization (wait, notify)
  - **short wait** ➔ use exclusion synchronization (synchronized)
- Confusion
  - exclusion used where condition required
  - synchronized (f) { // f is a Forklift instance
  - …} // spins until f is unlocked, arbitrary access

Pitfall #6:
Confusing long and short waits (2/5)



- synchronized (f) {
- while (busy) wait();
- busy = true;
- // Operate the forklift
- busy = false;
- notify();
- }
- The wait loop has no effect!
  - The thread that sets busy to true also keeps object f locked so no other job that needs forklift can reach the wait loop.

Pitfall #6:

Confusing long and short waits (3/5) (a possible solution)

- f.acquire();
- // use f
- f.release();

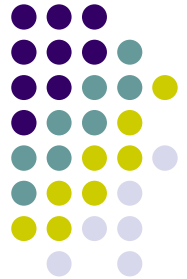
```
synchronized void acquire() {
  while (busy) wait();
  busy = true;
}
        synchronized void release() {
          busy = false;
          notify();
        }
```

Pitfall #6:
Confusing long and short waits (4/5) (nested critical sections)

- class V … {
- synchronized void m( ) {
- synchronized (wo) {
- while (cond) wo.wait( ) ;
- } } }
- if cond is true, the calling thread
  - enters wo's wait set,
  - releases lock on wo,
  - but keeps vo (instance of V) locked, and other threads needing vo spins, instead of waiting in wait set

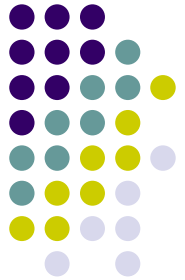
Pitfall #6:

Confusing long and short waits (5/5) (nested critical sections)



- class V … {
- synchronized void m( ) {
- synchronized (wo) {
- while (cond) wait( );
- } } }
- If vo is current instance of V and cond is true, the calling thread
  - enters vo's wait set, and
  - releases its locks on vo,
  - while keeping wo locked!

Strategy #6:
Confusing long and short waits



- Understand the **difference** between
  - exclusion synchronization (synchronized)
  - condition synchronization (wait, notify)
- Which to use?
  - no threads → no need of exclusion sync
  - → remaining constraint use condition sync
  - eg: allocate forklift to one job at a time in sequential implementation too → use condition synchronization
- Tool can flag any wait calls in statically nested critical sections