Java Threaded Programming

Zhuang Guo
Paul Lu

Constructing threads

- Use constructors in the Thread class:
  - *Thread()*

Start/stop a thread

- Call *start()* method
- Then, a new thread will be started to execute *run()*.
- Method *isAlive()* returns true when *run()* is executing.
- Static method *Thread.currentThread()* returns the current running thread.
- A thread stops when the *run()* method returns.

Put a thread to sleep

- Call *sleep(long millis)* or *sleep(long millis, int nanos)*
- A sleeping thread can be interrupted. When that happens, an *InterruptedException* object is thrown.
- Therefore, always sandwich *sleep(long millis)* in a try-catch block.

Wait a thread to stop

- Use methods:
  - *join()*

Synchronization Mechanisms

- Java uses synchronized methods or synchronized statements to wrap critical code sections.
- Each critical code section is monitor-guarded. A lock is acquired and released when enter or exit a critical code section.
- In Java, each object has a lock. Class is also an object.
- Use static method *Thread.holdsLock(Object o)* to check if the current thread holds the object's lock.

Synchronization mechanisms cont'd

- Synchronized statements

```
synchronized( object A){
  // At entrance, acquire A's lock
  //Thread.holdsLock(A) returns true.
  // Execute statements…
  …
  // At exit, release A's lock
}
```

Synchronization mechanisms cont'd

- Synchronized non-static methods of object A:

```
synchronized void method(…){
  // At entrance, acquire A's lock
  //Thread.holdsLock(A) returns true.
  execute method statements.
  …..
  // At exit, release A's lock.
}
```

Synchronization mechanisms cont'd

- Synchronized static methods of object A:

```
synchronized static void method(…){
  // At entrance, acquire A's class lock
  execute method statements.
  …..
  // At exit, release A's class lock.
}
```

Thread scheduling

- Java multithreading is preemptive.
- Use *getPriority()* and *setPriority(int priority).* Their effects varies across different platforms.
- Use *yield()* to let other threads of equal priority to execute.

Wait/notify()

- Method w*ait()* waits until being notified or interrupted.
- Method *wait(long millis)* only waits for a specified amount of time.
- Method *notify()* notifies one waiting thread.
- Method *notifyAll()* notifies all waiting threads.
- Note: acquire the object's lock first before issue *wait/notify* on the object.

Thread interruption

- Interrupt a thread using the *interrupt()* method.
- When interrupted, an *InterruptedException* object is thrown, and the sleeping/waiting thread resumes execution.
- Method *isInterrupted()* returns true when a thread is interrupted.
- Static method *interrupted()* also returns true when interrupted, but it also clears the flag.

Thread groups

- Each thread belongs to one thread group.
- Each thread group contains threads and other thread groups.
- The root thread group is the *system* group.
- Thread's method activeCount() and enumerate(Thread[] theArray) returns the number of active threads in the calling thread's group or subgroup and lists these active threads.

Thread groups cont'd

- ThreadGroup's *activeGroupCount()* method returns the estimated number of active thread groups.
- A thread's maximal priority in a thread group can be set by method *setMaxPriority(int priority).*
- ThreadGroup's *interrupt()* interrupt all the threads in the group and subgroups.

Timer

- Use *Timer* and *TimerTask* to schedule the execution of one task at a specified time.
- A thread is associated with a *Timer* object.
- Methods in Timer:
  - One time execution:
    void schedule(TimerTask task, Date time)
  - Mutliple times:
    void schedule(TimerTask task, Date firstTime, long interval)

One more thing

- Do not call deprecated methods in *Thread,* such as suspend(), resume(), because they can lock up your program and damage objects.

Thread local variable cont'd

- A ThreadLocal field should be declared static, since it makes no sense to let each thread to have its own copy.
- The InheritableThreadLocal class allows a child thread to inherit the thread local variable of each parent.

Thread local variables

- The *ThreadLocal* class allows each thread to have its own thread-local storage.
- *ThreadLocal* class has three major methods:
  - *Object get()*: gets the thread's local value.
  - *Object intialValue()*: returns a thread local variable's initial value. This method's default implementation returns null, so it must be subclassed.
  - *void set(Object value)*: set the thread's local value.