Multi-Processing in Java

Fred Long

Reading

- "Java Thread", (second edition), by Scott Oaks and Henry Wong, O'Reilly & Associates, Inc., 1999, ISBN 1-56592-418-5.

Java Threads

- Java allows the creation of many threads
- Threads run concurrently
- A thread is created by instantiating an object of a class that either
  - extends the **`Thread`** class, or
  - implements the **`Runnable`** interface

Thread Methods

- **`void run()`** — the method that the newly created thread will execute.
- **`void start()`** — creates a new thread and executes the **`run()`** method defined in this thread class.

```
public class MyClass extends Thread
{
   public void run()
   {
      // do something interesting
   }
}

...
MyClass mc = new MyClass();
mc.start();
...
```



```
public class MyClass implements Runnable
{
   public void run()
   {
      // do something interesting
   }
}

...
Runnable mc = new MyClass();
Thread th = new Thread(mc);
th.start();
...
```


Other Thread Methods

- **`static void sleep(long milliseconds)`**
  `static void sleep(long millis, int nanos)`
  these put the current thread to sleep for the specified number of milliseconds (and nanoseconds)
- **`boolean isAlive()`** — determines if a thread is considered alive.


```
public class MyClass implements Runnable {
  Thread timer;
  public void start() {
    if (timer == null) {
      timer = new Thread(this);
      timer.start();
    }
  }
  public void run() {
    while (isAlive()) {
      // do something interesting
      Thread.sleep(1000);
    }
    timer = null;
  }
}
```

Joining Threads

- **`void join()`**
  `void join(long timeout)`
  `void join(long timeout, int nanos)`

  the current thread waits for the thread to which join is applied to complete, but no longer than the timeout.

Synchronizing Threads

- Threads may be synchronized on objects by using the **`synchronized`** keyword
- **`synchronized`** may be applied to a whole method, or to a block
- Every object has a mutex lock associated with it; if a synchronized method wishes to access the object, its executing thread must grab the lock before it can continue


```
private float total;
public synchronized boolean deduct
  (float amount)
{
  if (amount <= total) {
    total -= amount;
    return true;
  }
  return false;
}
---
synchronized(total) {
  // do something with total
}
```


Thread Communication

- **`void wait()`** — causes a thread to wait for a condition; it must be called from within a synchronized method or block
- **`void notify()`** — communicates to a thread that is waiting for a condition that the condition has occurred
- **`void notifyAll()`** — communicates to all threads waiting on the object

Implementing a Semaphore

```
public class BusyFlag {
  protected Thread busyflag = null;
  protected int busycount = 0;
  public synchronized void getBusyFlag();
  public synchronized boolean
    tryGetBusyFlag();
  public synchronized void freeBusyFlag();
  public synchronized Thread
    getBusyFlagOwner();
}
```



```
public synchronized void getBusyFlag() {
  while (!tryGetBusyFlag()) {
    try {
      wait();
    } catch (Exception e) {}
  }
}
```

```
public synchronized Thread
                    getBusyFlagOwner() {
  return busyflag;
}
```



```
public synchronized boolean
                    tryGetBusyFlag() {
  if (busyflag == null) {
    busyflag = Thread.currentThread();
    busycount = 1;
    return true;
  }
  if (busyflag==Thread.currentThread()) {
    busycount++;
    return true;
  }
  return false;
}
```



```
public synchronized void freeBusyFlag() {
  if (getBusyFlagOwner ==
      Thread.currentThread()) {
    busycount--;
    if (busycount == 0) {
      busyflag = null;
      notify();
    }
  }
}
```

Thread Interruption

- **`void interrupt()`** — sends an interruption to the specified thread
- **`static boolean interrupted()`** — determines whether a thread has been interrupted
- **`boolean isInterrupted()`** — determines whether the specified thread has been interrupted

Deprecated Methods

- **`void stop()`** — terminates a running thread
- **`void suspend()`** — prevents a thread from running
- **`void resume()`** — allows a suspended thread to run

Java Thread Scheduling

- Every thread is in one of four states: initialising, runnable, blocked, exiting
- the JVM is responsible for scheduling the runnable threads
- **`void setPriority(int priority)`** — allows a thread's priority to be set (thread priorities must be in the range 1–10)

Priority Inversion

- If a high priority thread attempts to acquire a lock held by a low priority thread it temporarily runs with an effective priority of the low priority thread
- This is solved by *priority inheritance* — the low priority thread holding the lock is temporarily given the priority of the high priority thread

Scheduling Implementations

- Green Threads — handled by the JVM, the running thread continues until blocked, usually uses priority inheritance
- Windows Native Threads — threads scheduled by the OS, only 7 priorities, priority inheritance, round-robin
- Solaris Native Threads — uses lightweight processes which timeslice but threads per LWP do not timeslice, uses priority inheritance

The Event-Dispatching Thread

- When the JVM executes, it starts a thread
- When the first AWT (or JFC) related class is instantiated, one or more additional threads are created
- One of these is responsible for getting events (key presses, mouse movements and clicks, etc.) from the user, and dispatching them

JFC Objects

- All JFC objects are thread-unsafe
- So, our own threads should not invoke methods on JFC objects directly
- Our thread must arrange for the event-dispatching thread to pass back the data
- Also, we should not attempt to synchronize on JFC objects

Thread Groups

- Java allows threads to be grouped together, using the ThreadGroup class
- All threads within a group may be operated on together
- Also, a thread hierarchy can be built, and this is the basis for Java’s thread security policy