CS108, Stanford
Fall, 2008-09

Handout #24
Osvaldo Jiménez

Thread 2 Interruption

Thanks to Nick Parlante for much of this handout

Interruption

- Interruption is about stopping the run of thread `A` with a command from thread `B`. This interaction is always a little random, as thread `A` could be at any point, halfway through some operation, at the moment that thread `B` decides to interrupt. Later on, we will use interruption to implement a *Stop* button that halts a bunch of worker threads.

worker.interrupt()

- Send to a thread object to signal that it should stop, e.g. `worker.interrupt()` interrupts the run of the thread object pointed to by "worker".
- Interruption does not stop the thread right away. There are two ways the thread can learn that it has been interrupted:
- a. In most cases, `interrupt()` sets an "interrupted" `boolean` in the thread to `true`.
- b. Or, if the thread is blocked in a `sleep()`, `join()`, or file reading operation, it gets popped out of that operation by an `InterruptedException`, but the interrupted `boolean` is not set. This exception mechanism is necessary, since if the thread is not running, it cannot check actively check its interrupted `boolean` (it's asleep!) -- `InterruptedException` takes care of those cases.
- The interrupted thread should notice, eventually, that it has been interrupted in one of those two ways, and exit its run loop cleanly.
- Because there is a delay between when someone sets interrupted on a thread, and the thread itself notices that it has been interrupted, we say that `interrupt()` is "asynchronous" -- `interrupt()` returns to its caller immediately, and the actual exit of the worker happens sometime after that.

worker.isInterrupted() -- Check Interrupted Bit

- `boolean isInterrupted();` in the `Thread` class
- Call `isInterrupted()` on a thread to check if it has been interrupted.
- Typically, a worker thread object sends this message to itself in its run loop periodically to see if it has been interrupted.
- When interrupted, the worker should exit its run, leaving its data structures in a clean state.
- `boolean interrupted()` -- very similar to `isInterrupted()`, but clears the flag -- do not use.

isInterrupted() vs. InterruptedException -- Exclusive

- When a thread is interrupted, it will be informed either by having its `isInterrupted boolean` set, or if the thread is blocked in `sleep()`, `join()`, etc. it will receive an `InterruptedException`, however a thread does not get both forms of notification. The thread gets one or the other. Therefore, `isInterrupted()` will return `false` if the thread was notified via an `InterruptedException`.

Old stop()/synchronous style

- Java used to feature "synchronous" thread control methods, including a `stop()` method, that affected the thread immediately. The synchronous approach has been deprecated because it is practically impossible to achieve the "exits leaving the data structures in clean state" condition in a program with synchronous thread control. A thread could get stopped partway through a statement, and so leave a data structure in a half-updated state in a way that makes it impossible for the program to continue reliably. e.g. Suppose code was part way through adding a new link to a doubly-linked list. For this reason the whole synchronous "stop" style is being phased out (not just in Java)

StopWorker Example

```
/*
 Demonstrates creating a couple worker threads, running them,
 interrupting them, and waiting for them to finish.
 In run(), increments a counter, prints something,
 and sleeps. Checks for interruption on each iteration.
*/
public class StopWorker extends Thread {

    public void run() {
        long sum = 0;
        int count = 500;
        int i;
        for (i=0; i<count; i++) {
            sum = sum + i; // do some work
            System.out.println(getName() + " " + i);

            // 1. Check interrupted boolean -> break
            if (isInterrupted()) {
                // clean up, exit when interrupted
                // (getName() returns a default name for each thread)
                System.out.println(getName() + " interrupted");
                break;
            }

            // 2. Sleep a little (simulate doing something slow)
            // InterruptedException -> break
            try {
                Thread.sleep(1);
            }
            catch (InterruptedException ex) {
                break;
            }

            // We notice we are interrupted either
            // because isInterrupted() is true or because
            // we get an InterruptedException.
        }

        // Notice if we exited the loop due to interruption.
        if (i < count) {
            System.out.println(getName() + " interrupted " + i);
        }
    }

    public static void main(String[] args) {
        StopWorker a = new StopWorker();
        StopWorker b = new StopWorker();

        System.out.println("Starting...");
        a.start();
        b.start();

        try {
            Thread.sleep(100); // sleep a little, so they make some progress
        }
        catch (InterruptedException ignored) {}

        System.out.println("Sending interrupt()");
        a.interrupt();
        b.interrupt();

        try {
            a.join();
            b.join();
        }
        catch (InterruptedException ignored) {
            // could get here if someone interrupted the main() thread
        }
        System.out.println("All done");
    }
/*
   Starting...
    Thread-0 0
    Thread-1 0
    Thread-0 1
    Thread-1 1
    Thread-0 2
    Thread-1 2
    Thread-0 3
    ...
    Thread-1 116
    Thread-0 117
    Thread-1 117
    Sending interrupt()
    Thread-0 interrupted 117
    Thread-1 118
    Thread-1 interrupted 118
    All done
*/
}
```