Lecture 03: Thread API (continue)

SSC2
Behzad Bordbar
School of Computer Science, University of Birmingham, UK
Recap

- Extending Thread or implementing Runnable
- Thread terminology
- Stopping Threads
- What is the current running thread?
- Stopping threads
Contents

- Stopping Threads
- Sleep
- Interrupt
- multiple threads
Stopping a thread (Recap)

- The only safe way to stop a thread is for it to stop itself.
- A thread \( t_1 \) can interrupt running of a thread \( t_2 \) by invoking its `interrupt()` method:

  ```java
  public void interrupt()
  ```

- Call `interrupt()` on a thread to signal that it should stop itself. The effect of this is to "set the interrupt status" of the thread, i.e. sets a flag in the destination thread indicating it has been terminated and returns right away.
Stopping a thread (Recap)

- The `run` method can check the interrupt status by calling `Thread.interrupted()`.

```java
public static boolean interrupted()
```

- This returns a boolean result to say whether or not the interrupt status (of the currently executing thread) is set.

- At the same time, it also resets the interrupt status so calling `Thread.interrupted()` a second time will return `false` (unless there has meanwhile been yet another call of `interrupt()`).
Why sleep throw this exception?

- To make sure the thread knows if `interrupt()` is called on it.

- If a thread is asleep, it cannot be calling `Thread.interrupted()` to check the interrupt status. Therefore

1) If the current thread has its interrupt status set when it tries to call `Thread.sleep()`, then it is not allowed to go to sleep - it stays runnable.
2) If a thread is asleep when interrupt() is called on it, then it is immediately woken up - it becomes runnable again.

- In each case, an InterruptedException is thrown and the interrupt status is reset. (It does not need to be set any more, because the InterruptedException means the thread now knows about the interrupt.)
A thread is *interrupted* if some other thread calls `interrupt()` on it.

Normally, when a thread is interrupted it should try to stop itself (by finishing its run method).

There are two ways a thread can find out it has been interrupted: either by calling `interrupted()`, or by getting an `InterruptedException`.
InterruptedException in run method

Example:
The printer CoolPrinter which reads files from a PrintSpooler (a queue of documents). The CoolPrinter has two mode of operation the *normal mode* and *repair mode*. ServicingManager periodically stops the CoolPrinter for checking and servicing. Create coolPrinter and run in on a thread. Let servicingManger interrupt it for the repair. ((see coolPrinter folder))
public void run(){
    boolean isInRepairMode = false;
    while(!isInRepairMode){
        ...
        try {
            KeepPrinting();
            ...
        } catch (InterruptedException e) {
            isInRepairMode = true;
            Reparing();
        }
    }
}
This pattern, known as *Guarded Block*, with a boolean variable to control stopping, is very common in properly written run methods, and you should learn it.

Note how we don't need to call `interrupted()` any more. Calling `isInRepairMode` on every repetition allows the method to find out if the interrupt status is set.
When you call `sleep` in methods other than `run`, quite often you don't know how to stop the thread - because you don't know what threads might call the method.

In the situation where the `InterruptedException` arises, ask yourself, *Can the method still perform its usual task?*

If not, then the method should throw an exception - that is what exceptions are for. Usually you just declare the method as throwing `InterruptedException`, and then you don't need a try-catch round `Thread.sleep()`.
InterruptedException in other methods

- If the method can successfully perform its usual task, despite the InterruptedException, then it is usually more convenient not to rethrow the exception.

- If you don't, you must make sure the interrupt status is set again. This is so that when control returns to the run method it can detect the interrupt.
InterruptedException in other methods

Use the following code.

```java
try {
    Thread.sleep(...);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
```

Note: The static method `Thread.currentThread()` returns a reference to the thread currently executing.
Deprecation methods and \texttt{stop()}

- Thread class has a method \texttt{stop()} that forces the thread to stop executing
- \texttt{stop()} is deprecated as it is unsafe. While a thread is updating an object, it may temporarily put the object in a bad state with its invariants broken.
- There is no safe way to force a thread to stop.
Many Thread in action

- Threads consume a lot of system resources:
  - Each thread uses memory
  - Creating and destroying threads takes a lot of time.
- This can be a big overhead if you have lots of small tasks that need to be run on separate threads.
- Therefore it is desirable to be able to limit the total number of threads, and to maximize the use of each thread once it has been created.
Example: Multi-threaded client/server

Thread 1 generates data

Thread 2 makes requests to server

Receipt & queuing

Requests

Input-output

Server pool (N threads)

Client

Server
Thread pools

- A thread pool keeps a fixed set of threads that are available for running a succession of short tasks,
- Each thread in the pool is presented as a run-object (i.e. an instance of a class that implements Runnable)
- Think of the threads in the pool as being like counter staff in a post office, and the run-objects as being like customers, each with its own task to perform.
- The threads take on any tasks that come along, and when all the threads are busy the tasks wait in a queue.
Thread pools via java.util.concurrent

A thread pool with nThreads threads in it, use this static factory method from java.util.concurrent.Executors:

```java
public static ExecutorService newFixedThreadPool(int nThreads)
```

which results in a Thread pool. Then you can call many methods, for example:

```java
void execute(Runnable command)
```

"command" is task to be submitted to the pool
Thread pools via
java.util.concurrent (continue)

If any threads are available to run it, one of them does so. Otherwise the task waits in a queue until a thread is free. Throws RejectedExecutionException if the thread is shut down.

```java
void shutdown() shuts the pool down. No new tasks will be accepted, though tasks still being run, or in the queue, will be completed.
```
Example: 3 waiters in a busy café

Four groups of customers have just arrived in a restaurant, sitting on four tables. There are two waiters. Each table spend up to 10 min (OK! to speed things 10 sec) to decide and order their food. Using Thread pool write a program that simulates the two waiters serving a four tables.

((see the folder restaurant for a solution))