Cancelling Java Threads

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Killing Threads

- You likely expect for Java's Thread class to have some `stop()`, or `kill()`, or `cancel()` method
- **It does not!**
- And yet, we need to be able to forcefully terminate threads all the time
  - e.g., In our GUIs, we'll want some "Cancel" buttons that stop ongoing stuff
    - And ongoing stuff is all in threads due to our JavaFX Golden Rule from the previous set of lecture notes
- So how do we do this????
- This will be our first incursion into "much more in depth than ICS332" content
  - Many, many more are coming…

Deferred Cancellations

- Java *used to* have a **`Thread.stop()`** method, but it has been deprecated for a long time
- Why would such a useful method be deprecated?
- Because it's too dangerous!
- What if the thread is in the middle of doing something like copying records from one data structure to another?
  - If you brutally terminate it, then the thread's job will be, say, half done, which likely will break the whole application
- So Java cannot allow programmers to simply kill threads because they might not do it right
  - The typical Java philosophy, which has a lot of merit, even though it often frustrates some developers
- Great, but what do we do then?

Deferred cancellations

- The point of deprecating the **`Thread.stop()`** method is that a thread shouldn't be stopped willy-nilly at any point of its execution
- This means that the implementation of a thread must specify when this thread can be stopped safely
- In other terms: during its execution, a thread should check regularly whether it should stop or not
- This is called deferred cancellation, and can be done easily with a variable
- It's called this way because the thread will terminate (possibly quite a bit) after it's been told to terminate
- Let's see this on an example…

Deferred Cancellation Example

Victim

```
public class Victim extends Thread {
  private boolean shouldStop = false;

  public void tellToStop() {
    this.shouldStop = true;
  }

  public void run() {
    System.out.println("Hello World!");
    while(true) {
      // check if I should stop
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
  }
}
```

Killer

```
. . .
// create the victim thread
Victim victim = new Victim();
victim.start();

try {
  Thread.sleep(1000);
} catch (InterruptedException e) {}

// tell the thread to stop
victim.tellToStop();

// Wait for victim
// to really have stopped
try {
  victim.join();
} catch (InterruptedException e) {}

. . .
```

- Program **`DeferredCancellation.java`** on course Web site

Deferred Cancellation Example

Victim

```
public class Victim extends Thread {
  private boolean shouldStop = false;

  public void tellToStop() {
    this.shouldStop = true;
  }

  public void run() {
    System.out.println("Hello World!");
    while(true) {
      // check if I should stop
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
  }
}
```

Killer

```
. . .
// create the victim thread
Victim victim = new Victim();
victim.start();

try {
  Thread.sleep(1000);
} catch (InterruptedException e) {}

// tell the thread to stop
victim.tellToStop();

// Wait for victim
// to really have stopped
try {
  victim.join();
} catch (InterruptedException e) {}

. . .
```

- Let's run it and see if it works….

Deferred Cancellation Example

Victim

```
public class Victim extends Thread {
  private boolean shouldStop = false;

  public void tellToStop() {
    this.shouldStop = true;
  }

  public void run() {
    System.out.println("Hello World!");
    while(true) {
      // check if I should stop
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
  }
}
```

Killer

```
. . .
// create the victim thread
Victim victim = new Victim();
victim.start();

try {
  Thread.sleep(1000);
} catch (InterruptedException e) {}

// tell the thread to stop
victim.tellToStop();

// Wait for victim
// to really have stopped
try {
  victim.join();
} catch (InterruptedException e) {}

. . .
```

- It doesn't!!!

Deferred Cancellation Example

Victim

```
public class Victim extends Thread {

  private volatile boolean
shouldStop = false;

  public void tellToStop() {
    this.shouldStop = true;
  }

  public void run() {
    System.out.println("Hello World!");
    while(true) {
      // check if I should stop
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
  }
}
```

Killer

```
. . .
// create the victim thread
Victim victim = new Victim();
victim.start();

try {
  Thread.sleep(1000);
} catch (InterruptedException e) {}

// tell the thread to stop
victim.tellToStop();

// Wait for victim
// to really have stopped
try {
  victim.join();
} catch (InterruptedException e) {}

. . .
```

- This fixes it!! (anybody's used **`volatile`** before?)

The volatile keyword in Java

- The story about **`volatile`** is interesting, useful, but a bit long
- So for these lecture notes, let's just accept that it works
  - If a thread doesn't "see" a variable, make it volatile
- The next set of lecture notes explains all mysteries associated with **`volatile`**
- Let's resume talking about thread deferred cancelations…

Deferred Cancellation

- Summary so far:
  - We cannot forcefully kill a thread
  - Instead, we can set some variable and the thread can check, whenever it's safe, whether it should terminate based on the variable value
- **One remaining question:** What if the thread is blocked waiting for something, and thus will not run the code that checks the variable value?
  - e.g., it's sleeping, it's waiting for a network connection,…
- Let's see another example

Blocked Thread Example

- Say that in your application you have a thread, A, that's currently waiting on something, say, an incoming network connection
- The main thread realizes that that network connection will never come in (due to whatever application logic)
- So the main thread "terminates" thread A by setting the relevant **`volatile`** variable to true
- But thread A is not checking the variable's value
  - It is stuck on some network call!!
- Therefore, it will never terminate
- It is very bad form to leave "zombie" threads like that lying around
  - And in fact it could break some applications' logic

How to Cancel a Blocked Thread

- What we need: a way to make the thread "snap out" of whatever is blocking it
  - Calls like **`sleep()`**, **`join()`**, **`wait()`**, etc.
- To do this, we use the **`Thread.interrupt()`** method
- Calling **`interrupt()`** causes an **`InterruptedException`** to be raised in the target thread's code if/while it is blocked
- Let's see an example, where we want to cancel a thread that calls **`sleep()`**…

Example of Thread.interrupt()

Victim Thread

```
public class Victim extends Thread {
  private volatile boolean die = false;

  public void stop() {
    this.die = true;
  }

  public void run() {
    . . .
    if (die) return;
    . . .
    try {
      Thread.sleep(100000000);
    } catch (InterruptedException e) {
      if (die) return;
    }
    . . .
}
```

Killer

```
Victim = new Victim();
victim.start();

. . .
// kill victim
victim.stop();
victim.interrupt();
```

- If `interrupt()` is called while the victim is not sleeping, nothing happens
- But if the victim is sleeping, then it will snap out of it and check the variable

Thread.interrupt() Thoughts

- So now we (finally?) understand the use of the **`InterruptedException`** exception
- When you write your own code, if you know that you will not interrupt your threads (or you only have one thread!), then you typically do nothing when the exception is caught
  - This is why I have, in a lot of code I've shown so far in the course, empty catch clauses
- But if you write code with others and are in charge of the code of one of many threads, it's always a good idea to deal with the **`InterruptedException`**
- Note that you can call **`Thread.interrupt()`** for whatever purpose, not necessarily thread cancellation
  - To force the thread to do something "now", and then perhaps resume its blocking/waiting operation

This is it

- This is it for cancellation of Java threads!
  - Who knew it could be so involved?
    - And we haven't even explained `volatile`
  - But this is concurrency for you (it's hard, but at least it's hopefully interesting...)
- Of course, many people need this, and so Java provides a ready-made solution: java.util.concurrent.FutureTask
  - We'll talk about java.util.concurrent later
  - Our goal here is to truly understand the low level before using all kinds of high-level, opaque abstractions
- We'll have to talk about suspending/resuming threads as well
  - But before we get there, we'll have to talk about more advanced topics

Conclusion

- We can now start on our multi-assignment project
- This project will expose you to all main aspects of concurrency
- It will also entail non-trivial, real software development project
  - Be prepared for “software redesign” and “code refactoring” sessions
  - Going for “quick and dirty” will be deadly
  - Talking with others about software design will be key
- With this in mind, let’s look at Homework #2….