Cannellation and Shutdown

Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

March 18, 2018
Task cancellation
  Cancellation with blocking operations
  Interruption policies
  Dealing with non-interruptible blocking
  Cancellable Task Factory

Stopping a thread based service
  Poison pills
  Limitations of shutdownNow

Abnormal termination
  Uncaught Exception Handlers

JVM shutdown
  Ghosts in town
  Finalizers

Cancellation and Shutdown Summary
Cancellation and Shutdown

- While it is easy to create tasks and threads, **stopping** them in a *proper* and *controlled* way is less trivial.
- Java does not provide a *safe* method to *stop* a thread. See http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html
- Instead you should use interruption, a cooperative mechanism that lets one thread ask another to stop what it is doing.
- This cooperative approach is fundamental since only the **task inside the interrupted thread** can know how to leave the shared data structures in a proper state.
- On interruption, a task should clean up and then terminate (return from its run() method).
Thread cancellation is about cooperation...

**Murder** is a messy business.

- Someone has to clean up.
- So why not let the victim do it; Ask it to cleanup and then to commit suicide...
- Suicide is painless anyway.
Cancellable

An activity is cancellable if *external* code can move it to completion before its normal completion.

There are a number of reasons for cancellation

- User requested cancellation
- Time-limited activities
- Application Events
- Errors
- Shutdown
Cooperative processing of cancellation

Cancellation requires cooperation

Thread.stop and friends leave a Thread in a not so nice state. They almost surely leave the objects in an invalid state (invariant might not hold and such things).

Therefore, a cancellation should be handled and programmed as a request to the task, that uses some kind of flag to post the request and regularly interrogates this flag in its task processing loop. Note that such a flag will be set in a different thread and therefore must be volatile.
Cancellation and Shutdown

Task cancellation
Cancellation with blocking operations
Cancellation of interruptible threads
Dealing with non-interruptible blocking
Cancellable Task Factory

Stopping a thread
Based service
Poison pills
Limitations of shutdownNow

Abnormal termination
Uncaught Exception Handlers

JVM shutdown
Ghosts in town
Finalizers

Cancellation and Shutdown
Summary
Cancellation invoked Listing 7.2, Calling Task.cancel();

```java
static List<BigInteger> aSecondOfPrimes() throws InterruptedException {
    PrimeGenerator generator = new PrimeGenerator();
    exec.execute(generator);
    try {
        SECONDS.sleep(1);
    } finally {
        generator.cancel();
    }
    return generator.get();
}

public static void main(String [] args){
    try {
        List<BigInteger> list = aSecondOfPrimes();
        System.out.println("a second of primes" + list);
    } catch (Throwable t){}
}
```
Thread interrupt related interface

The interface that `java.lang.Thread` provides for interruption.

```java
public class Thread implements Runnable {

    public void interrupt() {...}

    public boolean isInterrupted() {...}

    public static boolean interrupted() {...}

    // and clears the interrupted state of the current Thread

    Quiz: These methods have only effect when a Thread is alive. When is a Thread alive?
```
Interruption

- The cancellation in the previous slides only works if the task sees the flag set. While the thread is waiting in a blocking operation (e.g. sleep) it will not see this.
- There always is the possibility that the flag is never seen after a certain point. (For instance when putting on a full queue.)
- In the API there are certain blocking operations that support interruption.

Interruption is the way to go

In fact the book says: Using interruption for anything else but cancellation is fragile and difficult to sustain in larger applications.
Broken PrimeProducer ... Question Time!

class BrokenPrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    private volatile boolean cancelled = false;

    BrokenPrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!cancelled)
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
        }
    }

    public void cancel() {
        cancelled = true;
    }
}
The effect of a Interrupt

Calling `interrupt()` does not stop the thread from doing what it is doing; It just delivers the message, which is saved in the boolean interrupted flag.

This request can be considered at cancellation points. Some methods take this request seriously (like: `wait()`, `sleep(int millis)`, and `join()`) and will throw a `InterruptedException`.

Interruption is the most sensible way to implement cancellation. In fact the needed flag is already available and reacted upon by the above mentioned methods.
public class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;

    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /* Allow thread to exit */
        }
    }

    public void cancel() {
        interrupt();
    }
}
Interrupt Policies

- Just as *Tasks* should have an cancellation policy, threads should have an *interruption* policy.
- An interruption policy determines how the thread interprets the interruption request.

The most sensible interruption policy is some kind of thread- or service level cancellation with:

- exit as quickly as possible
- cleaning up if needed and
- possibly notify the thread-owner

It is important to make a difference between *tasks* and *threads* in their way to react on interrupts. An interrupt may mean different things:

- Cancel current task or
- shutdown worker thread and
- may have more than one desired recipient.
Tasks and interrupts I

Tasks do not execute in threads of their own; they borrow the thread which is owned by some service like a **ThreadPool**.

Code that does not own a thread should be careful to preserve the interrupted status so that the thread-owning service may react to it.

The guest task may react upon an interrupt, but must not hide the signal from the owner. This can simply be implemented by (re)throwing an **InterruptedException** like most blocking library methods do.
Tasks and interrupts II

A task is not (always) obligated to drop all work once interrupted. It may remember the interrupted status, finish what it was doing and then throw an `InterruptedException` or otherwise inform its caller on a more opportune moment. This policy can protect data structures from corruption (invariants!).

A task should not make any assumptions on the policies of its executing thread (thread policy is not its responsibility), so if it does act on an `InterruptedException`, it must preserve the interrupted status to its host-thread by calling `Thread.currentThread().interrupt()`.
Tasks and interrupts III

Just as task code, cancellation code must not make any assumptions either on the interrupt policy of an unknown thread. A thread should only be interrupted by its owner (the thread pool for instance).

Each thread may have its own interruption policy

STOP So stay off.
Because each thread has its own interruption policy, you should not interrupt a thread, unless you know (for instance because you are the owner of the thread) what interruption means to that thread.
Interrupt response

When calling a method that may throw an `InterruptedException`, there may be two strategies to deal with it:

- Propagate the exception (possibly after some cleanup), which makes your method into an exception throwing method too, or
- Restore the interruption status so the code high up on the call stack can deal with it.

Only code that implements the thread’s interruption policy may swallow an interruption request. General-purpose tasks and library code should never swallow interruption requests.
Uncancellable activities and interrupts

If an activity does not support cancellation but does call interruptable blocking methods, it must do so in a loop, retrying when an interrupt is detected. In that case the interrupt status should be saved locally and passed to the caller by restoring it just before returning. See next slide.
Preserving interrupt example Listing 7.7

```
public Task getNextTask(BlockingQueue<Task> queue) {
    boolean interrupted = false;
    try {
        while (true) {
            try {
                return queue.take();
            } catch (InterruptedException e) {
                interrupted = true;
                // fall through and retry
            }
        }
    } finally {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```
public class TimedRun1 {
    private static final ScheduledExecutorService cancelExec = Executors.newScheduledThreadPool(1);

    public static void timedRun(Runnable r, long timeout, TimeUnit unit) {
        final Thread taskThread = Thread.currentThread();
        cancelExec.schedule(new Runnable() {
            public void run() {
                taskThread.interrupt();
            }
        }, timeout, unit);
        r.run();
    }
}

Do not do this (Note our mr. Yuk!); you have no control where the interrupt will fire with respect to the task’s execution.
Timed Run, 2nd Attempt, part 1

```java
class TimedRun2 {
    private static final ScheduledExecutorService cancelExec = newScheduledThreadPool(1);

    public static void timedRun(final Runnable r, long timeout, TimeUnit unit)
            throws InterruptedException {
        class RethrowableTask implements Runnable {
            private volatile Throwable t;

            public void run() {
                try {
                    r.run();
                } catch (Throwable t) {
                    this.t = t;
                }
            }

            void rethrow() {
                if (t != null) throw launderThrowable(t);
            }
        }
```

Summary
Timed Run, 2nd Attempt, part 2

```java
RethrowableTask task = new RethrowableTask();
final Thread taskThread = new Thread(task);
taskThread.start();
cancelExec.schedule(new Runnable() {
    public void run() {
        taskThread.interrupt();
    }
}, timeout, unit);
taskThread.join(unit.toMillis(timeout));
task.rethrow();
```

Here, when the execution was interrupted, that information is retrieved after the join and returned to the caller (as a thrown, laundered, Exception). But the problem is in the join; it does have a return value (it is void), so the caller cannot distinguish between success full join (Runnable terminated normally) or a timed out join.
Reason to cancel

Some times the reason will announce it selves.

"Cancel the meeting on our Five-Year Planning, Ms. Duncan."

© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com
Timed Run with Future

```java
public class TimedRun {
    private static final ExecutorService taskExec = Executors.newCachedThreadPool();

    public static void timedRun(Runnable r, long timeout, TimeUnit unit)
            throws InterruptedException {
        Future<?> futureTask = taskExec.submit(r);
        try {
            futureTask.get(timeout, unit);
        } catch (TimeoutException e) {
            // task will be cancelled below
        } catch (ExecutionException e) {
            // exception thrown in task; rethrow
            throw launderThrowable(e.getCause());
        } finally {
            // Harmless if task already completed
            futureTask.cancel(true); // interrupt if running
        }
    }
}
```

Future has **and** takes the responsibility to preserve the interrupted state and forward it to the caller as part of the returned result.
Cancel the future

Cancel a future if you no longer need the result

When `Future.get()` throws (a postponed) `InterruptedException` or `TimeoutException` and you know that the result is no longer of use to the program, cancel the task with `Future.cancel()`.
Blocking library methods that are non-interruptible

Some of the IO methods in the library do not respond to interruption. In this case there are other ways to get their attention:

- Closing the socket or channel prematurely. Synchronous socket and (interruptible) channel IO react by throwing a exception (SocketException and AsynchronousCloseException respectively)
- `close()` on a Selector also has the effect of throwing and interrupt on the next wake up call. (In the book: add close to the text on page 148).
- In implicit locks (of the `wait()` and `notify()` kind) you are out of luck, the only way it sees the interrupted state of its thread is when it can make some progress (by being waked with `notify()`)
- When using explicit `Locks`, you can use the `lockInterruptibly` to lock and still be responsive to interrupts
**Nonstandard cancellation: overriding interrupt in Thread**

```java
public class ReaderThread extends Thread {
    private static final int BUFSZ = 512;
    private final Socket socket;
    private final InputStream in;

    public ReaderThread(Socket socket) throws IOException {
        this.socket = socket;
        this.in = socket.getInputStream();
    }

    @Override
    public void interrupt() {
        try {
            socket.close();
        } catch (IOException ignored) {
        } finally {
            super.interrupt();
        }
    }

    public void run() {
        // By extending Thread and then override interrupt() you can use interrupt() to close the Reader and pass the interrupt up.
    }
}
```

By extending `Thread` and then override `interrupt()` you can use `interrupt()` to close the Reader and pass the interrupt up.
A pattern in action. Note the direction of the arrows and the absence of dependency cycles.
Code of SocketUsingTask class

Listing 7.12

```java
public abstract class SocketUsingTask<T> implements CancellableTask<T> {
    @GuardedBy("this") private Socket socket;

    protected synchronized void setSocket(Socket s) {
        socket = s;
    }

    public synchronized void cancel() {
        try {
            if (socket != null)
                socket.close();
        } catch (IOException ignored) {
        }
    }

    public RunnableFuture<T> newTask() {
        return new FutureTask<T>(this) {
            public boolean cancel(boolean mayInterruptIfRunning) {
                try {
                    SocketUsingTask.this.cancel();
                } finally {
                    return super.cancel(mayInterruptIfRunning);
                }
            }
        };
    }
}"
```
Interface and factory method

Listing 7.12, interface (== type)

```java
interface CancellableTask <T> extends Callable<T> {
    void cancel();

    RunnableFuture<T> newTask();
}
```

Factory method hook since Java 6, (two patterns at once)

```java
protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
    if (callable instanceof CancellableTask)
        return ((CancellableTask<T>) callable).newTask();
    else
        return super.newTaskFor(callable);
}
```
Problems in logging: example

Listing 7.13 deserves a 😞, because it may leave producers (the clients of this service) blocked in the bounded queue. The producers will never become unblocked again.

This is not a satisfactory shutdown technique.

Listing 7.14 tries to solve it, but it has a race condition (caused by a check, then act sequence), which still may cause producers to become blocked indefinitely.

The solution is in listing 7.15, which adds a reservation and can use this to make the check and act atomic.
Logging with executor

```java
public void start(){ }

public LogServiceUsingExecutor(File logFile) 
    throws FileNotFoundException{ 
    writer = new PrintWriter( logFile );
}

public void stop() throws InterruptedException { 
    try{ 
        exec.shutdown(); 
        exec.awaitTermination(TIMEOUT, UNIT);
    } finally { 
        writer.close();
    }
}

public void log( String msg ){
    try {
        exec.execute( new WriteTask(msg) );
    } catch ( RejectedExecutionException ignored ) { }
}
```

Quiz: write the WriteTask(String msg) class.
Using a special message in the queue

If your queue is purely FIFO, you can add a special message as a signal to stop a service.

- In case of one producer and one consumer, putting one pill in the queue suffices.
- In case of multiple consumers, each consumer gets a pill.
- In case of multiple producers, the consumer has to count and swallow the pills. This sounds 😞, but it isn’t (at least not from a programmer’s point of view).
- In case of multiple producers and consumers, you better get into the pharmaceutical industry 😎, as it comes down to $C \times P$. 😊.
- The queue must be unbound for this to reliably work.
Abrupt termination always complicates matters.

The standard way a ExecutorService implements shutdown is that it attempts to cancel the tasks currently in progress and returns a list of tasks that were submitted but never started.

It does not (cannot without the help of the tasks) know the state of the tasks that were in progress and their completion.

With the help of some *instrumentation* (wrapping in this case), an executor can be made to collect the tasks that were cancelled at shutdown. See next sheet.
Note that this implementation does a composition and delegation. (after the mantra *prefer composition over inheritance*). Returns task that were started at cancel time. The interesting methods are below.

```java
public List<Runnable> getCancelledTasks() {
    if (!exec.isTerminated())
        throw new IllegalStateException(/*...*/);
    return new ArrayList<>(tasksCancelledAtShutdown);
}
```

The executor wraps the runnable with a runnable catching the `isShutdown()`

```java
public void execute(final Runnable runnable) {
    exec.execute(new Runnable() {
        public void run() {
            try {
                runnable.run();
            } finally {
                if (isShutdown() && Thread.currentThread().isInterrupted())
                    tasksCancelledAtShutdown.add(runnable);
            }
        }
    });
}
```
The leading cause of premature thread dead is *cancer* `RuntimeException`.

`RuntimeException`s are generally not caught. Not catching them lets the exception boil up through the call stack and then terminate the causing thread. In a single threaded (e.g. console) application this kills your program.

If this termination is *benign* or *disastrous* fully depends on the thread’s role in the application. Losing one thread in a pool is no big deal, but if your GUI event thread would die, the user would surely notice.

Just as about any creature can catch cancer, just about any code can throw `RuntimeException`s. The threads in your ThreadPool pond act as guinea-pigs in that they call unknown code, which might cause them to die.
Threadpool worker structure Listing 7.23

The way to report death of a thread.

```java
public class ThreadPoolThread extends Thread {
    public void run() {
        Throwable thrown = null;
        try {
            while (!isInterrupted())
                runTask(getTaskFromWorkQueue());
        } catch (Throwable e) {
            thrown = e;
        } finally {
            threadExited(this, thrown);
        }
    }
}
```
UncaughtExceptionHandler

The Thread API provides hooks to handle uncaught exceptions. By implementing the `UncaughtExceptionHandler` interface, you can provide your own handler.

```java
public class UEHLogger implements Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e) {
        Logger logger = Logger.getLoggerAnonymousLogger();
        logger.log(Level.SEVERE, "Thread terminated with exception: ",
                   t.getName(), e);
    }
}
```

If you have a long-running application

In a long-running application, always use uncaught exception handlers for all threads that at least log the exception.
Uncaught Exception Handlers and ThreadPools

If you need an `UncaughtExceptionHandler` in a ThreadPool, provide a ThreadFactory to the ThreadPoolExecutor constructor. Note that this should be done by the thread’s owner only.

If you want to be notified (and you want that, or stay confused) either wrap the exception-catching `Runnable` or `Callable` or use the `afterExecute` hook in the `ThreadPoolExecutor`.

Note that for tasks submitted with submit, any exception belongs to the “result” or return value of that task. The exception is saved and re-thrown in the `Future.get()` method, after being wrapped in a `ExecutionException`.

Quiz: How do you get the real cause in the last case?
Kinds of shutdown

The JVM can shutdown in two ways: *orderly* or *abruptly*.

An orderly shutdown is initiated when a program ends in a normal way, that is when the last non-daemon thread terminates, someone calls `System.exit()` or by a platform specific way of shutdown like control-C or sending a INT signal with `kill`. These are the preferred ways.

Abnormal termination is achieved by calling `Runtime.halt` or by killing the JVM process with a more abrupt kill signal like `SIGKILL (9)` or control-\."
Shutdown hooks

In an abrupt shutdown the JVM is just halted. period. No cleanup etc.

In an orderly shutdown the JVM start the shutdown hook-threads. These threads are not yet started and the order in which they run is not guaranteed. After that, the JVM runs the finalizers if `runFinalizersOnExit` is true. If either of these threads or finalizers do not complete, the orderly shutdown process hangs, and you will still have to give a `kill -KILL` signal to the JVM process.

Shutdown hooks are threads, run concurrently, even to not yet stopped normal application threads and must therefor be programmed thread safely.

Sometimes it is better to use one shutdown hook for all services in your application. By implementing it with a shutdown task-queue, you can avoid concurrency problems there and control execution order.
Daemon Threads I

Threads are in two camps: Normal threads and Daemon threads.

Daemon threads are used as background services, the best example being the garbage collector.

The difference is only in the exit behavior: If the “house” inhabits only daemon threads, the JVM is stopped by means of an orderly shutdown. This happens when (the last) normal, non-daemon thread stops.
Daemon Threads II (in a haunt-ed house?)

Daemon threads are simply abandoned, just as you would a haunted house. So use daemons sparingly, because they may be abandoned abruptly. There are few tasks that can safely be abandoned at any time.

Certainly do not do any I/O. Just keep them for housekeeping chores, like putting out the garbage.

We are not in the ghost(-busters) department

Daemon threads are no substitute for a properly managed service life cycle within an application.
Avoid finalizers

The garbage collector does a good job of reclaiming memory resources. (In fact, being a garbage collector wins you praise on many a java conference, and there are many contenders. The quality of the garbage collector is often determining the overall quality of the JVM).

When putting out the garbage, GC (as we now affectionately call him) runs the non-trivial finalizers. They again must be threadsafe or at least access synchronized.

finalizers offer no guarantees of being run and impose a performance penalty if non-trivial. They are also extremely difficult to write correctly. See http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TSo281.pdf

You might want to follow Hans Boehms advice: you are allowed only one finalize()er per 10k lines.
Summary

End-of-life-cycle issues for tasks, threads, services and applications add complexity to design and implementation.

Java does not provide a preemptive mechanism for cancellation of activities or terminating threads.

*Interrupt* is a *cooperative* mechanism that can be used to facilitate cancellation, but it is up to the designer and implementer to construct cancellation protocols and use them consistently.

*FutureTask* and *Executor* simplify these issues.