CHAPTER 7

Cancellation and Shutdown

It is easy to start tasks and threads. Most of the time we allow them to decide when to stop by letting them run to completion. Sometimes, however, we want to stop tasks or threads earlier than they would on their own, perhaps because the user cancelled an operation or the application needs to shut down quickly.

Getting tasks and threads to stop safely, quickly, and reliably is not always easy. Java does not provide any mechanism for safely forcing a thread to stop what it is doing.[1] Instead, it provides *interruption*, a cooperative mechanism that lets one thread ask another to stop what it is doing.

The cooperative approach is required because we rarely want a task, thread, or service to stop *immediately*, since that could leave shared data structures in an inconsistent state. Instead, tasks and services can be coded so that, when requested, they clean up any work currently in progress and *then* terminate. This provides greater flexibility, since the task code itself is usually better able to assess the cleanup required than is the code requesting cancellation.

End-of-lifecycle issues can complicate the design and implementation of tasks, services, and applications, and this important element of program design is too often ignored. Dealing well with failure, shutdown, and cancellation is one of the characteristics that distinguishes a well-behaved application from one that merely works. This chapter addresses mechanisms for cancellation and interruption, and how to code tasks and services to be responsive to cancellation requests.

7.1 Task cancellation

An activity is *cancellable* if external code can move it to completion before its normal completion. There are a number of reasons why you might want to cancel an activity:

1. The deprecated `Thread.stop` and `suspend` methods were an attempt to provide such a mechanism, but were quickly realized to be seriously flawed and should be avoided. See `http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html` for an explanation of the problems with these methods.

User-requested cancellation. The user clicked on the "cancel" button in a GUI application, or requested cancellation through a management interface such as JMX (Java Management Extensions).

Time-limited activities. An application searches a problem space for a finite amount of time and chooses the best solution found within that time. When the timer expires, any tasks still searching are cancelled.

Application events. An application searches a problem space by decomposing it so that different tasks search different regions of the problem space. When one task finds a solution, all other tasks still searching are cancelled.

Errors. A web crawler searches for relevant pages, storing pages or summary data to disk. When a crawler task encounters an error (for example, the disk is full), other crawling tasks are cancelled, possibly recording their current state so that they can be restarted later.

Shutdown. When an application or service is shut down, something must be done about work that is currently being processed or queued for processing. In a graceful shutdown, tasks currently in progress might be allowed to complete; in a more immediate shutdown, currently executing tasks might be cancelled.

There is no safe way to preemptively stop a thread in Java, and therefore no safe way to preemptively stop a task. There are only cooperative mechanisms, by which the task and the code requesting cancellation follow an agreed-upon protocol.

One such cooperative mechanism is setting a "cancellation requested" flag that the task checks periodically; if it finds the flag set, the task terminates early. `PrimeGenerator` in Listing 7.1, which enumerates prime numbers until it is cancelled, illustrates this technique. The `cancel` method sets the `cancelled` flag, and the main loop polls this flag before searching for the next prime number. (For this to work reliably, `cancelled` must be `volatile`.)

Listing 7.2 shows a sample use of this class that lets the prime generator run for one second before cancelling it. The generator won't necessarily stop after exactly one second, since there may be some delay between the time that cancellation is requested and the time that the `run` loop next checks for cancellation. The `cancel` method is called from a `finally` block to ensure that the prime generator is cancelled even if the the call to `sleep` is interrupted. If `cancel` were not called, the prime-seeking thread would run forever, consuming CPU cycles and preventing the JVM from exiting.

A task that wants to be cancellable must have a *cancellation policy* that specifies the "how", "when", and "what" of cancellation—how other code can request cancellation, when the task checks whether cancellation has been requested, and what actions the task takes in response to a cancellation request.

Consider the real-world example of stopping payment on a check. Banks have rules about how to submit a stop-payment request, what responsiveness guarantees it makes in processing such requests, and what procedures it follows when

```
@ThreadSafe
public class PrimeGenerator implements Runnable {
    @GuardedBy("this")
    private final List<BigInteger> primes
            = new ArrayList<BigInteger>();
    private volatile boolean cancelled;

    public void run() {
        BigInteger p = BigInteger.ONE;
        while (!cancelled) {
            p = p.nextProbablePrime();
            synchronized (this) {
                primes.add(p);
            }
        }
    }

    public void cancel() { cancelled = true; }

    public synchronized List<BigInteger> get() {
        return new ArrayList<BigInteger>(primes);
    }
}
```

Listing 7.1. Using a `volatile` field to hold cancellation state.

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

Listing 7.2. Generating a second's worth of prime numbers.

payment is actually stopped (such as notifying the other bank involved in the transaction and assessing a fee against the payor's account). Taken together, these procedures and guarantees comprise the cancellation policy for check payment.

`PrimeGenerator` uses a simple cancellation policy: client code requests cancellation by calling `cancel`, `PrimeGenerator` checks for cancellation once per prime found and exits when it detects cancellation has been requested.

7.1.1 Interruption

The cancellation mechanism in `PrimeGenerator` will eventually cause the prime-seeking task to exit, but it might take a while. If, however, a task that uses this approach calls a blocking method such as `BlockingQueue.put`, we could have a more serious problem—the task might never check the cancellation flag and therefore might never terminate.

`BrokenPrimeProducer` in Listing 7.3 illustrates this problem. The producer thread generates primes and places them on a blocking queue. If the producer gets ahead of the consumer, the queue will fill up and `put` will block. What happens if the consumer tries to cancel the producer task while it is blocked in `put`? It can call `cancel` which will set the `cancelled` flag—but the producer will never check the flag because it will never emerge from the blocking `put` (because the consumer has stopped retrieving primes from the queue).

As we hinted in Chapter 5, certain blocking library methods support *interruption*. Thread interruption is a cooperative mechanism for a thread to signal another thread that it should, at its convenience and if it feels like it, stop what it is doing and do something else.

There is nothing in the API or language specification that ties interruption to any specific cancellation semantics, but in practice, using interruption for anything but cancellation is fragile and difficult to sustain in larger applications.

Each thread has a boolean *interrupted status*; interrupting a thread sets its interrupted status to true. `Thread` contains methods for interrupting a thread and querying the interrupted status of a thread, as shown in Listing 7.4. The `interrupt` method interrupts the target thread, and `isInterrupted` returns the interrupted status of the target thread. The poorly named static `interrupted` method *clears* the interrupted status of the current thread and returns its previous value; this is the only way to clear the interrupted status.

Blocking library methods like `Thread.sleep` and `Object.wait` try to detect when a thread has been interrupted and return early. They respond to interruption by clearing the interrupted status and throwing `InterruptedException`, indicating that the blocking operation completed early due to interruption. The JVM makes no guarantees on how quickly a blocking method will detect interruption, but in practice this happens reasonably quickly.

```
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; }
}

void consumePrimes() throws InterruptedException {
    BlockingQueue<BigInteger> primes = ...;
    BrokenPrimeProducer producer = new BrokenPrimeProducer(primes);
    producer.start();
    try {
        while (needMorePrimes())
            consume(primes.take());
    } finally {
        producer.cancel();
    }
}
```

LISTING 7.3. Unreliable cancellation that can leave producers stuck in a blocking operation. *Don't do this.*

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    public static boolean interrupted() { ... }
    ...
}
```

LISTING 7.4. Interruption methods in `Thread`.

If a thread is interrupted when it is *not* blocked, its interrupted status is set, and it is up to the activity being cancelled to poll the interrupted status to detect interruption. In this way interruption is "sticky"—if it doesn't trigger an `InterruptedException`, evidence of interruption persists until someone deliberately clears the interrupted status.

Calling `interrupt` does not necessarily stop the target thread from doing what it is doing; it merely delivers the message that interruption has been requested.

A good way to think about interruption is that it does not actually interrupt a running thread; it just *requests* that the thread interrupt itself at the next convenient opportunity. (These opportunities are called *cancellation points*.) Some methods, such as `wait`, `sleep`, and `join`, take such requests seriously, throwing an exception when they receive an interrupt request or encounter an already set interrupt status upon entry. Well behaved methods may totally ignore such requests so long as they leave the interruption request in place so that calling code can do something with it. Poorly behaved methods swallow the interrupt request, thus denying code further up the call stack the opportunity to act on it.

The static `interrupted` method should be used with caution, because it clears the current thread's interrupted status. If you call `interrupted` and it returns `true`, unless you are planning to swallow the interruption, you should do something with it—either throw `InterruptedException` or restore the interrupted status by calling `interrupt` again, as in Listing 5.10 on page 94.

`BrokenPrimeProducer` illustrates how custom cancellation mechanisms do not always interact well with blocking library methods. If you code your tasks to be responsive to interruption, you can use interruption as your cancellation mechanism and take advantage of the interruption support provided by many library classes.

Interruption is usually the most sensible way to implement cancellation.

`BrokenPrimeProducer` can be easily fixed (and simplified) by using interruption instead of a boolean flag to request cancellation, as shown in Listing 7.5. There are two points in each loop iteration where interruption may be detected: in the blocking `put` call, and by explicitly polling the interrupted status in the loop header. The explicit test is not strictly necessary here because of the blocking `put` call, but it makes `PrimeProducer` more responsive to interruption because it checks for interruption *before* starting the lengthy task of searching for a prime, rather than after. When calls to interruptible blocking methods are not frequent enough to deliver the desired responsiveness, explicitly testing the interrupted status can help.

```
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(); }
}
```

LISTING 7.5. Using interruption for cancellation.

7.1.2 Interruption policies

Just as tasks should have a cancellation policy, threads should have an *interruption policy*. An interruption policy determines how a thread interprets an interruption request—what it does (if anything) when one is detected, what units of work are considered atomic with respect to interruption, and how quickly it reacts to interruption.

The most sensible interruption policy is some form of thread-level or service-level cancellation: exit as quickly as practical, cleaning up if necessary, and possibly notifying some owning entity that the thread is exiting. It is possible to establish other interruption policies, such as pausing or resuming a service, but threads or thread pools with nonstandard interruption policies may need to be restricted to tasks that have been written with an awareness of the policy.

It is important to distinguish between how *tasks* and *threads* should react to interruption. A single interrupt request may have more than one desired recipient—interrupting a worker thread in a thread pool can mean both "cancel the current task" and "shut down the worker thread".

Tasks do not execute in threads they own; they borrow threads owned by a service such as a thread pool. Code that doesn't own the thread (for a thread pool, any code outside of the thread pool implementation) should be careful to preserve the interrupted status so that the owning code can eventually act on it, even if the "guest" code acts on the interruption as well. (If you are house-sitting for someone, you don't throw out the mail that comes while they're away—you save it and let them deal with it when they get back, even if you do read their magazines.)

This is why most blocking library methods simply throw `InterruptedException` in response to an interrupt. They will never execute in a thread they own, so they implement the most reasonable cancellation policy for task or library code: get out of the way as quickly as possible and communicate the interruption back to the caller so that code higher up on the call stack can take further action.

A task needn't necessarily drop everything when it detects an interruption request—it can choose to postpone it until a more opportune time by remembering that it was interrupted, finishing the task it was performing, and *then* throwing `InterruptedException` or otherwise indicating interruption. This technique can protect data structures from corruption when an activity is interrupted in the middle of an update.

A task should not assume anything about the interruption policy of its executing thread unless it is explicitly designed to run within a service that has a specific interruption policy. Whether a task interprets interruption as cancellation or takes some other action on interruption, it should take care to preserve the executing thread's interruption status. If it is not simply going to propagate `InterruptedException` to its caller, it should restore the interruption status after catching `InterruptedException`:

```
Thread.currentThread().interrupt();
```

Just as task code should not make assumptions about what interruption means to its executing thread, cancellation code should not make assumptions about the interruption policy of arbitrary threads. A thread should be interrupted only by its owner; the owner can encapsulate knowledge of the thread's interruption policy in an appropriate cancellation mechanism such as a shutdown method.

Because each thread has its own interruption policy, you should not interrupt a thread unless you know what interruption means to that thread.

Critics have derided the Java interruption facility because it does not provide a preemptive interruption capability and yet forces developers to handle `InterruptedException`. However, the ability to postpone an interruption request enables developers to craft flexible interruption policies that balance responsiveness and robustness as appropriate for the application.

7.1.3 Responding to interruption

As mentioned in Section 5.4, when you call an interruptible blocking method such as `Thread.sleep` or `BlockingQueue.put`, there are two practical strategies for handling `InterruptedException`:

- Propagate the exception (possibly after some task-specific cleanup), making your method an interruptible blocking method, too; or
- Restore the interruption status so that code higher up on the call stack can deal with it.

Propagating InterruptedException can be as easy as adding InterruptedException to the throws clause, as shown by getNextTask in Listing 7.6.

```
BlockingQueue<Task> queue;
...
public Task getNextTask() throws InterruptedException {
    return queue.take();
}
```

Listing 7.6. Propagating InterruptedException to callers.

If you don't want to or cannot propagate InterruptedException (perhaps because your task is defined by a Runnable), you need to find another way to preserve the interruption request. The standard way to do this is to restore the interrupted status by calling interrupt again. What you should *not* do is swallow the InterruptedException by catching it and doing nothing in the catch block, unless your code is actually implementing the interruption policy for a thread. PrimeProducer swallows the interrupt, but does so with the knowledge that the thread is about to terminate and that therefore there is no code higher up on the call stack that needs to know about the interruption. Most code does not know what thread it will run in and so should preserve the interrupted status.

Only code that implements a thread's interruption policy may swallow an interruption request. General-purpose task and library code should never swallow interruption requests.

Activities that do not support cancellation but still call interruptible blocking methods will have to call them in a loop, retrying when interruption is detected. In this case, they should save the interruption status locally and restore it just before returning, as shown in Listing 7.7, rather than immediately upon catching InterruptedException. Setting the interrupted status too early could result in an infinite loop, because most interruptible blocking methods check the interrupted status on entry and throw InterruptedException immediately if it is set. (Interruptible methods usually poll for interruption before blocking or doing any significant work, so as to be as responsive to interruption as possible.)

If your code does not call interruptible blocking methods, it can still be made responsive to interruption by polling the current thread's interrupted status throughout the task code. Choosing a polling frequency is a tradeoff between efficiency and responsiveness. If you have high responsiveness requirements, you cannot call potentially long-running methods that are not themselves responsive to interruption, potentially restricting your options for calling library code.

Cancellation can involve state other than the interruption status; interruption can be used to get the thread's attention, and information stored elsewhere by the interrupting thread can be used to provide further instructions for the interrupted thread. (Be sure to use synchronization when accessing that information.)

```
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();
    }
}
```

Listing 7.7. Noncancelable task that restores interruption before exit.

For example, when a worker thread owned by a ThreadPoolExecutor detects interruption, it checks whether the pool is being shut down. If so, it performs some pool cleanup before terminating; otherwise it may create a new thread to restore the thread pool to the desired size.

7.1.4 Example: timed run

Many problems can take forever to solve (e.g., enumerate all the prime numbers); for others, the answer might be found reasonably quickly but also might take forever. Being able to say "spend up to ten minutes looking for the answer" or "enumerate all the answers you can in ten minutes" can be useful in these situations.

The aSecondOfPrimes method in Listing 7.2 starts a PrimeGenerator and interrupts it after a second. While the PrimeGenerator might take somewhat longer than a second to stop, it will eventually notice the interrupt and stop, allowing the thread to terminate. But another aspect of executing a task is that you want to find out if the task throws an exception. If PrimeGenerator throws an unchecked exception before the timeout expires, it will probably go unnoticed, since the prime generator runs in a separate thread that does not explicitly handle exceptions.

Listing 7.8 shows an attempt at running an arbitrary Runnable for a given amount of time. It runs the task in the calling thread and schedules a cancellation task to interrupt it after a given time interval. This addresses the problem of unchecked exceptions thrown from the task, since they can then be caught by the caller of timedRun.

This is an appealingly simple approach, but it violates the rules: you should know a thread's interruption policy before interrupting it. Since timedRun can be called from an arbitrary thread, it cannot know the calling thread's interrup-

```
private static final ScheduledExecutorService cancelExec = ...;

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();
}
```

Listing 7.8. Scheduling an interrupt on a borrowed thread. *Don't do this.*

tion policy. If the task completes before the timeout, the cancellation task that interrupts the thread in which `timedRun` was called could go off *after* `timedRun` has returned to its caller. We don't know what code will be running when that happens, but the result won't be good. (It is possible but surprisingly tricky to eliminate this risk by using the `ScheduledFuture` returned by `schedule` to cancel the cancellation task.)

Further, if the task is not responsive to interruption, `timedRun` will not return until the task finishes, which may be long after the desired timeout (or even not at all). A timed run service that doesn't return after the specified time is likely to be irritating to its callers.

Listing 7.9 addresses the exception-handling problem of `aSecondOfPrimes` and the problems with the previous attempt. The thread created to run the task can have its own execution policy, and even if the task doesn't respond to the interrupt, the timed run method can still return to its caller. After starting the task thread, `timedRun` executes a timed `join` with the newly created thread. After `join` returns, it checks if an exception was thrown from the task and if so, rethrows it in the thread calling `timedRun`. The saved `Throwable` is shared between the two threads, and so is declared `volatile` to safely publish it from the task thread to the `timedRun` thread.

This version addresses the problems in the previous examples, but because it relies on a timed `join`, it shares a deficiency with `join`: we don't know if control was returned because the thread exited normally or because the `join` timed out.[2]

7.1.5 Cancellation via Future

We've already used an abstraction for managing the lifecycle of a task, dealing with exceptions, and facilitating cancellation—`Future`. Following the general principle that it is better to use existing library classes than to roll your own, let's build `timedRun` using `Future` and the task execution framework.

2. This is a flaw in the `Thread` API, because whether or not the `join` completes successfully has memory visibility consequences in the Java Memory Model, but `join` does not return a status indicating whether it was successful.

```
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);
        }
    }

    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();
}
```

Listing 7.9. Interrupting a task in a dedicated thread.

`ExecutorService.submit` returns a `Future` describing the task. `Future` has a `cancel` method that takes a boolean argument, `mayInterruptIfRunning`, and returns a value indicating whether the cancellation attempt was successful. (This tells you only whether it was able to deliver the interruption, not whether the task detected and acted on it.) When `mayInterruptIfRunning` is `true` and the task is currently running in some thread, then that thread is interrupted. Setting this argument to `false` means "don't run this task if it hasn't started yet", and should be used for tasks that are not designed to handle interruption.

Since you shouldn't interrupt a thread unless you know its interruption policy, when is it OK to call `cancel` with an argument of `true`? The task execution threads created by the standard `Executor` implementations implement an interruption policy that lets tasks be cancelled using interruption, so it is safe to set `mayInterruptIfRunning` when cancelling tasks through their `Future`s when they are running in a standard `Executor`. You should not interrupt a pool thread directly when attempting to cancel a task, because you won't know what task is running when the interrupt request is delivered—do this only through the task's `Future`. This is yet another reason to code tasks to treat interruption as a cancellation request: then they can be cancelled through their `Future`s.

Listing 7.10 shows a version of timedRun that submits the task to an ExecutorService and retrieves the result with a timed Future.get. If get terminates with a TimeoutException, the task is cancelled via its Future. (To simplify coding, this version calls Future.cancel unconditionally in a finally block, taking advantage of the fact that cancelling a completed task has no effect.) If the underlying computation throws an exception prior to cancellation, it is rethrown from timedRun, which is the most convenient way for the caller to deal with the exception. Listing 7.10 also illustrates another good practice: cancelling tasks whose result is no longer needed. (This technique was also used in Listing 6.13 on page 128 and Listing 6.16 on page 132.)

```
public static void timedRun(Runnable r,
                            long timeout, TimeUnit unit)
                            throws InterruptedException {
    Future<?> task = taskExec.submit(r);
    try {
        task.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
        task.cancel(true); // interrupt if running
    }
}
```

Listing 7.10. Cancelling a task using Future.

When Future.get throws InterruptedException or TimeoutException and you know that the result is no longer needed by the program, cancel the task with Future.cancel.

7.1.6 Dealing with non-interruptible blocking

Many blocking library methods respond to interruption by returning early and throwing InterruptedException, which makes it easier to build tasks that are responsive to cancellation. However, not all blocking methods or blocking mechanisms are responsive to interruption; if a thread is blocked performing synchronous socket I/O or waiting to acquire an intrinsic lock, interruption has no effect other than setting the thread's interrupted status. We can sometimes convince threads blocked in noninterruptible activities to stop by means similar to interruption, but this requires greater awareness of why the thread is blocked.