Task cancelation

Why:

- user-requested cancelation
- time-limited
- application events
- errors
- shutdown

No way to **preemptively** stop a thread => no way to **preemptively** stop a task. There are only **cooperative** mechanisms: the task and the code requesting cancellation follow an agreed-upon protocol. One such cooperative mechanism is setting a “cancellation requested” flag.

Interruption

Mechanism

- each thread has a boolean interrupred flag
- interrupting a thread sets this flag to true
- **Thread** contains methods for interrupted
- **Thread** contains methods for querying the interrupted status of a thread: **isInterrupted** and **interrupted** (clears the status)

Blocking library methods like **Thread.sleep** and **Object.wait**:

- try to detect when a thread has been interrupted
- try to return early
- clear interrupted status
- throw InterruptedException

Preserving the cancellation status

2 ways to know about interruption:

- **interrupted** flag
- **InterruptedException**

Whatever way you get notified, you must propagate interruption higher up the stack because:

- cancellation policy
- thread interruption policy

Interruption policy and cancellation policy

A task that wants to be cancellable must have a **cancellation policy**:

- how other code can request cancellation
- when the task checks wether cancellation has been requested
- what actions the task takes in response to a cancellation request

Interruption policy consists of

- what it does (if anything) when one is noticed
- what units of work are considered atomic with respect to interruption
- how quickly it reacts to interruption

Most sensible interruption policy

- exit as quickly as possible
- clean up if necessary
- possibly notify some owning entity that the thread is exiting

Threads with nonstandard interruption policies may need to be restricted to tasks written with awareness of this policy

Tasks that do not own the thread should be careful to preserve the interrupted status so that the owning code can eventually act on it. Must distinguish between

- cancel the current task
- shut down the worker thread

A task should not assume anything about the interruption policy of its executing thread (unless it is designed to run within a service with specific interruption policy). Task should care to preserve the executing thread's interruption status:

- throw InterruptedException
- restore interrupt flag

A thread should be interrupted only by its owner, should make no assumptions about task cancellation policy.

*Cancellation via **Future***

ExecutorService.submit returns a **Future** that has a **cancel** method that takes a boolean argument **mayInterruptIfRunning**. When it is **True** and the task is running in some thread then the thread is interrupted. Task execution threads created by the standard Executor implementation implement an interruption policy that lets tasks be cancelled via interruption.

Dealing with non-interruptible blocking

Not all blocking methods are responsible to interruptions:

- Synchronous socket IO in **java.io**. **read** and **write** methods in **InputStream** and **OutputStream** are not responsive to interruption, but closing the socket makes all threads executing these methods throw a **SocketException**.

- Synchronous IO in **java.nio**. Interrupting a thread waiting on an **InterruptibleChannel** causes it to throw **ClosedByInterruptException**, closes the channel and causes other threads blocked on this channel to throw **ClosedByInterruptExcetpion**.
- Asynchronous IO with selector. If a thread blocked in **Select.select** (in **java.nio** channels) wakeup causes it to return prematurely by throwing a **ClosedSelectorException**.
- **Lock** acquisition. Not possible to interrupt thread waiting for an intrinsic lock but to wait it to acquier the lock and make some progress to notice the interrupt. **Lock** class offers the **lockInterruptibly** method for this reason.

Stopping a Thread-based Service

ExecutorService shutdown

- **shutdown** method, safe
- **shutdownNow** method, responsive, returns all tasks from queue not yet started

Poison Pills

With a FIFO queues posion pills mean

- producers don't send any data after poison pill
- consumers finish all the work prior the pill

Work only when number of consumers and producers is known

- Can be extended to **n** producers: consumer must receive pills from all producers
- Can be extended to **n** consumers: producer must send pills for all consumers

*Limitations of **shutdownNow***

shutdownNow terminates running tasks and returns list of submitted but not yet started tasks. There is no way to find out what tasks were terminated in progress.
Probably **shutdownNow** does not return tasks interrupted in progress because that would produce "false positives": tasks that actually completed but were terminated before quiting. Considering these tasks terminated would be ok if they were idempotent.

Handling abnormal thread termination

Approach 1

In task processing threads, catch *RuntimeException*s or add **final** block to take corrective actions

Approach 2

Using **UncaughtExceptionHandler**

- Set with **Thread.setUncaughtExceptionHandler** or with **Thread.setDefaultUncaughtExceptionHandler**
- Order of finding a handler: per-thread handler, thread-group handler, it's parents' group handler, … , top-level thread group that delegates to the default system handler (if one exists) or otherwise prints stacktrace to console.
- Log the message, or try to restart thread or shutdown an application or any other appropriate action

For pool threads:

- Provide a **ThreadFactory** to the **ThreadPoolExecutor** constructor.
- Standard thread pools allow an uncaught task exception to terminate the pool thread, use a try-finally block to replace the thread. That makes an illusion of tasks failing silently.

When exceptions are propagated

- only when tasks are submitted with **execute**
- for **submit** exceptions are part if return status

JVM shutdown

Shutdown

- in orderly manner: last non-daemon thread terminates, someone calls **System.exit**, other platform specifif means (**SIGING**)
- in abrupt manner: calling **Runtime.halt**, killing JVM process by the operating system

Shutdown hooks

Hooks are registered with **Runtime.addShutdownHook**

In an orderly shutdown

- JVM makes no guarantees on the order in which shutdown hooks are started
- any application thread (daemon or nondaemon) will run concurrently
- when shutdown hooks are completed JVM may choose to run finalizers if **runFinalizersOnExit** is true and then halts
- all threads are abrupty terminated when the JVM halts
- if the shutdown hooks or finalizers don't complete then orderly shutdown process "hangs" and JVM must be shut down abruptly

In an abrupt shutdown JVM is not required to do anything but to halt the JVM.

It could be a single hook

- to ensure they are executed in the right order
- to ensure thread safety and liveness

- to ensure they are execued in the right order

Shutdown hooks design pricinples

- should be thread safe
- should avoid deadlocks
- should make no assumptions about the state of application
- should make no assumptions about why JVM is shutting down
- should exit as fast as possible
- should not rely on services that can be shut down by the application or other hooks.

Hooks should be used for service or application cleanup.

Daemon threads

When JVM starts it creates daemon threads (gc, housekeeping) and normal thread **main**. When a new thread is created in inherits the daemon status of the thread that created it.

When a thread exits the JVM performs an inventory of running threads and if the only threads running that are left are daemon threds it initiates and orderly shutdown.

Finalizers

For reclaiming (such as file or socket handlers) must be explicitly returned to operating system. GC treats objects with nontrivial **finalize** methods specially, after they are reclaimed by the collector, **finalize** is called.

- must be thread safe
- no guarantee when and if they run
- impose a significant performance cost on objects with nontrivial finalizers
- extremely difficult to write correctly
- in most cases combination of **fainally** block with **close** invokation does a better job
- sole exception is when you need to manage objects that hold resources acquired by native methods