Callables, Futures and Thread Pools

Radu Nicolescu
Department of Computer Science
University of Auckland

15 May 2013

Recall the Runnable interface — run()

```
public interface Runnable
```

A task that can be executed by a thread and *does not* return any result.

Requires a single method:

```
void run()
```

What if we want to execute a thread on a task which *does* return a result (such as SwingWorker's doInBackground?)

The answer is ... via the Callable interface (next slide)

Callable<V> interface — call()

```
public interface Callable<V>
```

Requires a single method:

```
V call() throws Exception
```

Method call computes a result of type V, or throws an exception, if unable to complete.

Thus, Callable<V> is a value-returning task, which

- can be executed asynchronously (i.e. by a separate thread),
- returns a result of type V or otherwise throws an exception
  - which will be returned when you call get() (next slides)

Callable<V> example

An example, using an anonymous class implementing the Callable interface:

```
Callable<String> cs = new Callable<String>() {
  public String call() {
    String result = ... data crunching or search in database
    return result;
  });
```

We can assume that the data crunching or database search takes substantial time, so it makes real sense to create such a callable task.

Callable<V> interface

Problems:

- How can we pick the asynchronous result (i.e. the result obtained in another thread) and when?
- How do we pick call's exceptions, if any?
- How much control can we have on the executing thread?

The answer is ... via the Future interface or its implementations such as FutureTask (next slides)

Future<V> interface — get()

The Future<V> interface offers a complementary view on value-returning tasks, which focuses on the result and the ways to control its evaluation (from outside).

```
public interface Future<V>
```

A Future represents an asynchronous computation (i.e. computed in another thread) and its result, and offers several ways to control its execution.

V is the result type returned by this Future's get method

Future<V> methods

```
boolean cancel(...)

boolean isCancelled()

boolean isDone()

V get()
  throws ExecutionException,
         InterruptedException, CancellationException

V get(long timeout, TimeUnit unit)
  throws ExecutionException, TimeoutException
         InterruptedException, CancellationException
```

Future<V> methods

- cancel (...) : attempts to cancel the task (may not always succeed, in some special circumstances).
- isCancelled () : true, if the task was cancelled.
- isDone() : true, if the task has completed.
- ... (next slide)

Future<V> methods

- get() : waits, if necessary, for the computation to complete, and then picks its result, of type V; may throw exceptions:
  - CancellationException - if the computation was cancelled
  - ExecutionException - wraps the exception thrown by the call computation, if any
  - InterruptedException - if the current thread was interrupted (while waiting)
- get(time, timeunit) : as get(); may throw an additional exception:
  - TimeoutException - if this wait exceeds the given time limits

Future<V> implementations

- SwingWorker<V,T> (in next theme)
- FutureTask<V> (next slides)
- Both implement Runnable as well, so can be used, in several contexts, as "normal" threads.

FutureTask<V> class

```
public class FutureTask<V>
             implements Future<V>, Runnable, ... {

    public FutureTask(Callable<V> callable) { ... }

    ...
}
```

The FutureTask class is an implementation of the Future interface (and so is SwingWorker)

Problem: how to construct and how to execute a FutureTask?

FutureTask<V> example

- The mentioned FutureTask constructor creates a FutureTask associated to the given Callable.

```
Callable<String> cs = ... // as previously defined

FutureTask<String> fs =
                        new FutureTask<String>(cs);
```

- Alternatively, you can also obtain a FutureTask by submitting a Callable to a thread pool (next slides).

FutureTask<V> example run — call()+start()+get()

FutureTask implements Runnable, and can be started like any other runnable object.

```
Callable<String> cs = ... // as previously defined
FutureTask<String> fs = new FutureTask<String>(cs);

new Thread(fs).start();

try {
   String res = fs.get();
} catch (Exception ex) {
   ...
}
```

The call fs.get() will wait (i.e. block) until the future task fs completes...

Callable void?

Most API's support equally both classical void tasks (Runnable) and value-returning tasks (Callable).

However, there may be cases when you need to masquerade genuine void tasks as value-returning tasks, and you don't want to return artificial results (e.g., Integer)

Possible workaround: use Void, an uninstantiable type that has only one value, null

```
Callable<Void> cs = new Callable<Void>() {
  public Void call() {
    ... some data crunching
    return null;
  });
```

Quiz: can you make your own Void-like uninstantiable class?

Thread pools and executors

- One of the problems with threads: thread objects are resource hungry: they need long time to create and start, lot of memory to run, long time to cleanup and shutdown.
- The solution of this problem: use thread pools.
- A thread pool has a number of threads ready to run, which you can use, without paying much overhead costs.
- After completing your task, a thread pool thread is *not* destroyed, but *recycled* back into the thread pool.
- More than this, threads are not directly visible in your application. You interact with thread pools using so-called executors.

Thread pools and executors

Class Executors offers several factory methods to create a thread pool and is associated executor.

```
static ExecutorService newFixedThreadPool(int n)
       Creates a thread pool that reuses a fixed number of threads.

static ExecutorService newCachedThreadPool()
       Creates a thread pool that reuses its existing threads.
       It creates additional threads, if needed,
       and discards them, if not used for long periods.

static ExecutorService newSingleThreadExecutor()
       Creates an Executor that uses a single worker thread.
       Could be useful for debugging or ensuring task serialization.

...
```

Thread pools and executors

Example.

```
ExecutorService executor =
        Executors.newFixedThreadPool(10);
```

Creates a thread pool that reuses 10 threads.

Executor interfaces

There are three basic executor interfaces:

public interface Executor
supports starting Runnable tasks

public interface ExecutorService
extends Executor
adds support for Callable and Future tasks
and for task lifecycle

public interface ScheduledExecutorService
extends ExecutorService
adds support for future and periodic
task execution

Executor interface — execute(run)

```
public interface Executor {
        void execute(Runnable command)
        queues the command for execution
}
```

Example, instead of:

```
new Thread(fs).start();
```

use:

```
executor.execute(fs);
```

ExecutorService interface — submit(call), invokeAll(call...)

```
public interface ExecutorService {
        void execute(Runnable task)
        queues the task for execution, see Executor

        Future<T> submit(Callable<T> task)
        queues the value-returning task for execution
        returns a future representing the task's pending-result

        List<Future<T>> invokeAll(
              Collection<? extends Callable<T>> tasks)
                              throws InterruptedException
        executes all the given value-returning tasks
        automatic join, i.e., returns when all tasks have completed
        (normally or by exception)
        interruptible
        ...
}
```

ExecutorService interface

```
public interface ExecutorService {
        ...
        void shutdown()
        initiates an orderly thread pool shutdown
        previously submitted tasks are still executed
        but no new tasks are accepted

        boolean awaitTermination(
                        long timeout, TimeUnit unit)
                                throws InterruptedException
        blocks until all tasks have completed execution
        after a shutdown request, with timeout, interruptible

        List<Runnable> shutdownNow()
        initiates a forced thread pool shutdown (via interrupts)
        returns a list of tasks that never commenced execution
}
```

ExecutorService interface — call()+submit()+get()

An alternate version of the previous example.

```
Callable<String> cs = ... // as previously defined

FutureTask<String> fs = executor.submit(cs);

try {
   String res = fs.get();
} catch (Exception ex) {
   ...
}
```

fs.get() waits until the future task fs completes...

ExecutorService interface — call()...+invokeAll()

Another alternate version of the previous example.

```
Callable<String> cs = ... // as previously defined
Callable<String> ...

List<Callable<String>> tasks =
                    new ArrayList<Callable<String>>();
tasks.add(cs);
tasks.add(...);

List<Future<String>> strings =
                    executor.invokeAll(tasks);

... // process results by calling get() on each returned Future
```

executor.invokeAll(tasks) waits until all future tasks complete...
("poor man's fork-join", because not always the most efficient way)

ForkJoinPool thread pools

- For better performance, use a ForkJoinPool thread pool, which is specifically dedicated to support parallel computing of many fine grained computationally intensive tasks
- The ForkJoinPool tries to minimize the thread overhead and to ensure an even load balancing by task stealing
- Essentially, fork-join threads that finish earlier steal tasks from other still busy fork-join threads ☺

ForkJoinPool thread pools — here with invokeAll()

```
int procs =
        Runtime.getRuntime().availableProcessors();

int maxallowed_threads = procs; // or 2*procs?

ForkJoinPool fjexecutor =
        new ForkJoinPool(maxallowed_threads);

List<Future<String>> strings =
        fjexecutor.invokeAll(tasks);
```

A ForkJoinPool thread pool is created directly, not via Executors

The ForkJoinPool thread pool implements ExecutorServices, so you can call invokeAll() on it

Memento — how to work with

One void task:

1. implement Runnable.run()
2. call Thread.start() or ExecutorService.execute()
3. call Thread.join() (opt.)

One value-returning task:

1. implement Callable.call()
2. call ExecutorService.submit()
3. call Future.get()

One value-returning task:

1. implement Callable.call()
2. wrap into a FutureTask
3. call Thread.start()
4. call Future.get()

A list of value-returning tasks:

1. implement Callable.call()'s
2. create a List of Callables
3. call ForkJoinPool.invokeAll() or ExecutorService.invokeAll()
4. obtain a List of Futures
5. call Future.get() on each

Assignment hints

Consider a task, which can be anything between "light" (short execution time) or "heavy" (long execution time)

```
private int task(int i) {
    ...
    ...
    ...
}
```

How to efficiently parallelise a long sequence of such tasks?

For example, consider a sequence of size $= 1000000$ tasks

For simplicity, omit the further processing of tasks results

Assignment hints — sequential

```
for (int i = 0; i < size; i++) {
    int k = task(i);
}
```

Assignment hints — naive parallelisation

```
for (int i = 0; i < size; i++) {
    final int ii = i;
    callables.add(new Callable<Void>() {
        public Void call() {
            int k = task(ii);
            return null;
        }
    });
}

List<Future<Void>> futures =
        executor.invokeAll(callables);
```

Assume the right types for callables (List of Callable) and executor (ExecutorService)

Assignment hints — range parallelisation

```
int chunk = 1000; // problem and platform specific

for (int low = 0; low < size; low += chunk) {
    final int Low = low;
    final int High = Math.min(size, low+chunk);
    callables.add(new Callable<Void>() {
        public Void call() {
            for (int i = Low; i < High; i++) {
                int k = task(i);
            }
            return null;
        }
    });
}

List<Future<Void>> futures =
        executor.invokeAll(callables);
```

Assignment hints — contrast the two parallelisations

- In the naive parallelisation:
  - we set one callable for each task call
- In the range parallelisation:
  - we set one callable for a relatively long sequence of task calls
  - one callable covers (High-Low+1) calls

Assignment hints — which parallelisation?

- Unless all tasks are "heavy" or of widely varying "weights", range partitioning works best (esp. for many similarly "light" tasks)
- In fact, because of the parallelisation overhead, the naive parallelisation can even take more time than the sequential version
- Sample timings (on a machine with 4 logical cores)
  - sequential version: 0.685 secs
  - naive partitioning: 1.366 secs
  - range partitioning: 0.222 secs

Summary of admin costs (overheads)

- Threads cost a lot:
  - heavy costs: create, release (destroy)
  - medium costs: swap contexts
- Threads pools are better: rare creation/release costs
- Tasks (Runnable or Callable) cost much less, *individually*:
  - apparently minor costs: to enqueue/dequeue tasks
  - apparently minor costs: for calling run(), call() or get()
- However small, these apparently minor costs do sum up
- The performance depends on the ratio admin/business parts
- Grouping many finely grained actions into bigger actions helps