Java Candidate Test

==========

Implement a generic Worker mechanism that can be used by an application to perform various types of processing.

A Worker is an autonomous processing engine that uses its own processing resources (e.g. threads) to do its work. Each Worker should be able to process its own types of Work. Workers should perform their processing in an asynchronous way, NOT delaying in any way the hosting application's threads. Workers should operate in three states, INITIAL, OPERATIONAL and STOPPED as explained below:

- INITIAL: State right after Worker instantiation, NOT for processing Work.
- OPERATIONAL: State in which, Workers normally accept and process Work.
- STOPPED: State indicating possible imminent shutdown, NOT for processing Work.

The hosting application must be able to interact with Worker implementations via well defined interfaces. Specifically and at a minimum:

- An interface to control the life cycle of a Worker and to submit Work units.
- An interface or abstract class as the root of the Work unit hierarchy.

The interfaces should support Java Generics, to provide compile-time type safety to the hosting application's developers. The interfaces should be possible to be used in the following ways:

- Worker<SomeWork> worker = new SomeWorkWorker<SomeWork>();
- Attempting to submit SomeOtherWork to a Worker<SomeWork> should be reported as a compile-time error.

This needs clarifications..

do you actually mean you want to get an error during the “attempting to submit a wrong type of Work” or to “create an instance of SomeWorkWorker with a wrong Work type”?

If I understand correctly what you want to achieve:

```
class J {}
class J2 extends J {}
class J22 extends J2 {}
class J3 extends J {}
class J33 extends J3 {}
```

Commented [AG1]: So, as far as I understand, a single Worker is an engine.. e.g. an application will submit jobs through a single Worker instance (for this job type)?
worker.add(jobInstance1);
worker.add(jobInstance2);
?

Commented [AG2]: So… the Work (or Job) should inherit from some common Job/Work in order for Worker to be able to interact with it?

Commented [AG3]: You mean not the ‘Workers’ will be doing processing, but rather the “Works” (e.g. Jobs)?

Commented [AG4]: So what triggers going from “INITIAL” into “OPERATIONAL”? Since further it says that in the “INITIAL” there should be no hanging threads, then it looks like there is nothing to do or wait for anything after a Worker instantiation.. Am I missing something?

Commented [AG5]: Again not sure about the main trigger for the Worker to be going into this state. Is it some kind of external event? If a Worker have currently running threads and some other Jobs are waiting in a queue, what should happen in this state? Do you mean Worker should simply “not accept any other jobs” when it’s in this state?

Commented [AG6]: This is also not clear. Is it going to be an external even/call to shutdown and interrupt all currently executing jobs?

Commented [AG7]: See below

```
class W <T extends J> {
        public void add(T t) {};
}
class WWW2 <T extends J2> extends W<T> {
        public void add(T t) {};
}
class W2 <T extends J2> extends W<T> {
        public void add(T t) {};
}
class WW2 extends W<J2> {
        public void add(J2 t) {}
}
```

Then the following is sufficient and cuts the clutter:

```
W<J2> ww2 = new WW2();
```

vs.

```
W<J2> w2 = new W2<J2>();
```

Note: the last line gives an impression that you can do: new W2<J3>() .. You'll get a compile error of course, but that looks like a bad API design.

In other words: If you want to get a compile error during submission, then I don't understand the need for this SomeWorkWorker to be parameterized

The following state transitions should be supported: <NOT EXISTS> --> INITIAL --> OPERATIONAL <--> STOPPED

Workers should use their own threads and thread pools to perform their processing, according to the processing needs of each implementation. The only restrictions imposed regarding threads and thread pools are the following:

- When in the INITIAL or STOPPED states, a Worker should NOT have ANY threads alive.

- Transitioning into the OPERATIONAL state, a Worker should create and initialize its threads or thread pools.

- Transitioning from the OPERATIONAL to the STOPPED state, a Worker should ensure that when the hosting application's thread returns from the transition-triggering method, processing of ALL Work units being actively processed by the Worker's threads HAS FINISHED.

Workers should also make use of queueing mechanisms to both be able to accept Work units for processing even when all their processing resources are busy, and not impose any delays to the hosting application's threads. To prevent out-of-memory conditions, Workers should indicate their inability to further queue additional Work units, when their queues already have "many" items, by throwing an appropriate exception to the calling hosting application.

Commented [AG8]: Some kind of new state? Previously there were only 3

Commented [AG9]: So… it can go back to OPERATIONAL state?.. and how this relates to "possible imminent shutdown" ?

Commented [AG10]: What should happen when "hosting application"s all non-daemon threads finish?

Commented [AG11]: So what is the trigger for this transitioning?

Commented [AG12]: Which "thread" ? There may be multiple threads in thread pool of the Worker implementation.

Commented [AG13]: Not clear at all.

Commented [AG14]: Soo… once again.. let's say a Worker has few threads which run some Jobs.. and there are some Jobs in a queue waiting.. Why would a Worker suddenly decide that it needs to go into "STOPPED" state? Even after all Jobs had finished?

Commented [AG15]: Ok ok .. I've got it.. you simply want to have a bounded queue.

IMPLEMENTATION

To demonstrate use of the mechanism as well as to prove its correct operation, an example Worker should be implemented to use the TempConvert Web Service (http://www.w3schools.com/xml/tempconvert.asmx) to convert temperatures between Celcius and Fahrenheit. The temperature as well as the conversion to perform should be parameters of the corresponding Work unit. The Worker should perform no more than 5 concurrent calls to the Web Service, but also should not be limited to one, and it should be able to queue up to 20 Work units.

The demostrating program should create an instance of the Worker, submit a few Work units with various combinations of temperatures and conversions (demonstrating normal operation) and then attempt to saturate the queue by rapid submission of many Work units. It should then demonstrate state transitions and exit.

P.S.

Commented [AG16]: Or instantiated?

Commented [AG17]: So now I've got really confused. Is it the Worker who does the work, or a Work who does the actual work?

Commented [AG18]: So the Work or the Worker?

Commented [AG19]: *I still don't understand (if you have a hierarchy of Works e.g Jobs) why do you want to have both:*
a) hierarchy of Workers
b) have Workers be generic-based
if it would make sense rather to have either a) or b)