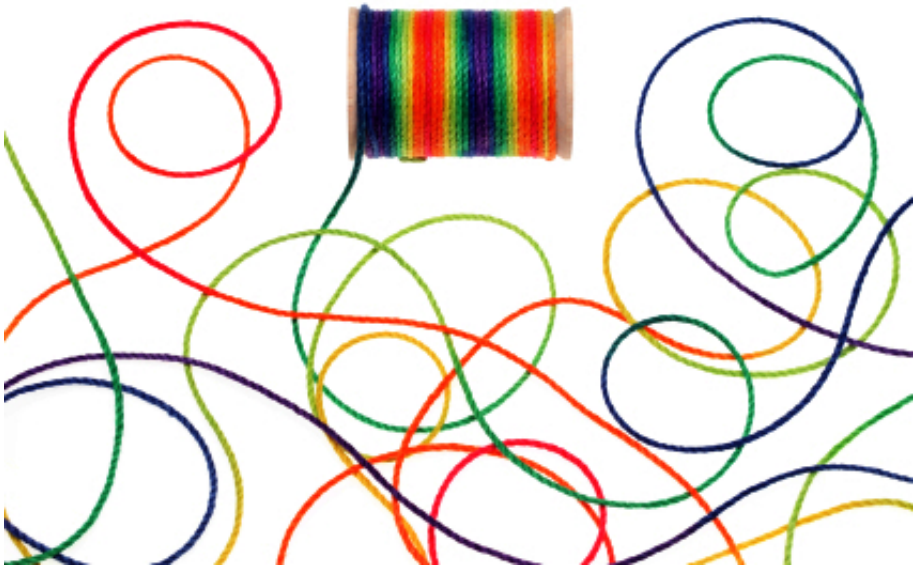
Java Active Objects

A Proposal

by Allen Holub



Allen does an end run around the JCP to give Java capabilities it's missing.

Multithreading is not for the faint of heart. Nevertheless, it's an essential part of server-side Java programming. The java.util.concurrent package went a long way to making multithreaded programs possible, but it didn't really solve the core problem: multithreaded programs are still way too complex, almost impossible to debug, and require you to have a deep knowledge of both multithreading theory and practice. Things don't have to be that difficult, however.

It turns out that one of the best solutions to the problem—one that's actually easy to do, without any of the complexity of a typical multithreaded system—is an architectural one. The design pattern in question is Active Object. The term was coined by Doug Schmidt in the mid nineties (for the ACE cross-platform threading framework), but the concept itself has been around much longer. In fact, it's built into many languages, ranging from Ada tasks (circa 1980) to Scala actors. The first time I saw the pattern was in Intel's Real-time Multitasking Executive embedded operating system (RMX), which also used the term "task." I'll use the word task as well.

A task, then, is an object that performs some sort of work for you asynchronously in the background. You ask it to do the work, and eventually, it does it. Many, if not most, multithreaded systems can be thought of in terms of tasks. For example, instead of executing a SQL query on a "worker thread," with all the concomitant synchronization problems, you can ask the database-update task to do the work for you in the background. If you care, the task can tell you when the work completes (or you can wait for it to complete). Tasks can talk to each other and, in fact, an entire program can be written as a set of cooperating tasks (that's what RMX did). This is a very different way of programming than many of us are used to, but it works well, and is way easier to program than standard solutions once you get used to it.

This article, then, is a modest proposal to add tasks to Java by means of annotations. An annotation-based solution is not optimal, since a few features would be better done by the compiler itself, but it's doable without having to convince Oracle to make changes to the language (which can take many years), and can be done as a simple open-source project without the Machiavellian political machinations of the JCP. This is very much a work in progress, and I'm sure I've missed whole swaths of issues, so I'd very much appreciate any comments and suggestions you may have; send me an email at jao@holub.com. I'll summarize the comments and post them to a JAO page on my website.

I've already implemented enough of an annotation-based task framework that I'm convinced it's workable, but I want some feedback before I proceed, which is my main motivation for writing about it here. I'm very interested in your comments. Once I've incorporated these, I'll actually open up the project for you to use. Again, send me an email and I'll tell you when/where to find the full implementation when it's available.

Also, as a heads up, I'm assuming in this article that you know a little about multithreading in general, java.util.concurrent in particular, and that you're at least superficially aware of the lambda-Expression extensions that will be part of Java 8. For the latter, pick up a beta copy of Venkat Subramaniam's Functional Programming in Java from the Pragmatic Bookshelf. You'll need to know this stuff anyway, and there's no time like the present to come up to speed.

Tasks

The best way to understand how to implement a task using Active Object is to look at one.

First, consider this class:

```
class NormalClass
{
    private double val = 0.0;
                                                                    //
    void doSomething()
    {   val = 1.0;
    }
                                                                    //
    void doSomethingElse()
    {   val = 2.0;
    }
}
```

The class is dangerous in a multithreading scenario because both methods can be called simultaneously, so the value of val (which is not atomic—it's updated in multiple steps) could be undefined—a classic race condition. You can, of course, use synchronization to solve this problem, which in this trivial case is easy. But once the class becomes realistically complex, synchronization can become very difficult.

Now let's do the earlier work using an Active-Object-based Task:

```
public class MyTask
{
    private double val = 0.0;
    private BlockingQueue<Runnable> dispatchQueue
        = new LinkedBlockingQueue<Runnable>();
                                                                    //
    public MyTask()
    {
        new Thread(
            new Runnable()
            {   @Override public void run()
                {   while(true)
                    {   try
                        {   dispatchQueue.take().run();
                        }
                        catch (InterruptedException e)
                        {   // okay, just terminate the dispatcher
                        }
                    }
                }
            }
        ).start();
```

```
    }
                                                                    //

    void doSomething() throws InterruptedException
    {   dispatchQueue.put(
            new Runnable()
            {   public void run(){ val = 1.0; };
            }
        );
    }
                                                                    //
    void doSomethingElse() throws InterruptedException
    {   dispatchQueue.put(
            new Runnable()
            {   public void run(){ val = 2.0; };
            }
        );
    }
}
```

Or, if you're using Java 8, you can simplify considerably using lambda expressions. The following code works just like the previous example (and I'll continue to use the lambda syntax from here on out):

```
public class MyTask
{
    private double val = 0.0;
    private BlockingQueue<Runnable> dispatchQueue =
        new LinkedBlockingQueue<Runnable>();
                                                                    //
    public MyTask()
    {
        new Thread( ()->
            {   while(true)
                {   try
                    {   dispatchQueue.take().run();
                    }
                    catch (InterruptedException e)
                    {   // okay, just terminate the dispatcher
                    }

                }
            }
        ).start();
```

```
    }
                                                                    //

    void doSomething() throws InterruptedException
    {   dispatchQueue.put( ()->{ val = 1.0; } );
    }
                                                                    //
    void doSomethingElse() throws InterruptedException
    {   dispatchQueue.put( ()->{ val = 2.0; } );
    }
}
```

The main issue, here, is that the Active-Object version is fully thread safe. No synchronization is required. You can program within the class as if the application were single-threaded.

The core of the Active Object pattern is a BlockingQueue and a single "dispatcher" thread. A blocking queue has two essential properties: (1) It's thread safe—multiple threads can enqueue concurrently, and (2) a dequeueing thread blocks until (waits for) an object to become available on the queue. The single dispatcher thread is idle until somebody enqueues a Runnable() object, at which time the thread dequeues the object, executes it, then goes back to waiting for the next request.

An Active Object, then, is effectively single threaded, even though it's used in a concurrent environment. Active Objects are concurrent relative to each other, but within the object, it's as if everything is single threaded. As a consequence, Active Objects can be written without worrying about threading issues, and so are much simpler.

The whole point of the exercise is the one I mentioned a few sentences back. There's absolutely no need for synchronization here (beyond the implicit synchronization in the BlockingQueue, of course.) Even if doSomething() and doSomethingElse() are called concurrently, the actual work (the assignments to val) are done serially by the dispatch thread, not concurrently.

In the Real World

Of course, in the real world, it's not that easy.

Active Objects assume that none of the objects that they use change unexpectedly, and that assurance can be tricky to get when you don't have the compiler helping you. For one thing, objects can come from outside (as method arguments, for example). Similarly, if you ever return a reference to an internal object, then another thread could be modifying that object while the Active Object uses it. Since the Active Object isn't synchronizing, this concurrent modification is a very real problem.

The best solution is to be very hard core when it comes to object orientation: In a pure OO system, objects are intelligent agents that do work for you in an opaque way. You make requests, the object does the work, and it gives you a response. You have no idea how it does its work.

A properly done object has to be rigid about basic principles like data encapsulation and implementation hiding. That is, it should simply not be possible to discover or impact how an object works from the outside. I think of it as using a rule I've eponymously dubbed the Holub Replacement Principle: it should be possible

to completely discard the implementation of a class (all method bodies, all fields, everything), replace the old implementation with a new one, and, as long as the interface hasn't changed, the class's clients—the set of objects that use the class—shouldn't notice. Well-structured objects like that are vastly easier to maintain than typical Java classes, which leak implementation details all over the place. For example, I've written extensively about how accessors and mutators (get/set functions) hurt maintainability, and they do that because they violate encapsulation. For example, if during the normal course of maintenance, you need to get rid of, or change the type of, or add an ancillary field to a field that's exposed using accessors, you're in deep trouble. Every call to the accessor is now broken. You can't fix this problem with an automated refactor.

Designing systems that don't have things like get/set methods in them is both possible and easy, once you get the hang of it (which, admittedly, can take some training), but at the core is the notion of delegation: ask for help, not for information. Put another way: don't ask some object for the information you need to do your work; rather, ask the object that has the information to do the work for you.

Which brings us back to tasks: A Task is an object that does something for you. It does that something in the background at its own pace, and can notify you when the work is done. You talk to it asynchronously (you make a request, but you don't get an immediate response, and typically don't wait for the response). Threads, then, are implicit in the notion of tasks, since a task typically does its work on a background thread.

When it comes to Tasks, this sort of hard-core encapsulation moves from a nice-to-have to a necessity. If you violate the encapsulation, the now-accessible internal fields become simultaneously accessible by multiple threads, and the whole design pattern breaks down and fails.

Looking back at the earlier example, if val were public, or if there was a getVal() and setVal() that accessed it, then we'd have absolutely no advantage in using the Active-Object pattern. That is, if a thread other than the dispatcher thread can access the local field, then we have to synchronize in the normal way. The whole point of the exercise is to eliminate the need for synchronization, however. Hard-core encapsulation is mandatory.

So, when possible, we want the compiler to enforce the rules for us. If we modified Java to force us to use good OO structure, we wouldn't be programming in Java anymore. However, we can use the annotation preprocessor to identify Active Object-based tasks and identify (or prohibit) coding problems that would break the Active Object.

Let the Compiler Do the Work For Us: A Basic @Task

Let's look at how that annotation processor works.

There are two problems with manually implementing Active Object: (1) It's annoying to do the boilerplate work of setting up the dispatch mechanism, and (2) it's difficult to guarantee that the active object is actually safe to use—there are no [*** Delete 'no'? ***] problems like unwanted external access to what should be guarded fields, for example. Both of these problems can be solved by adding a handful of annotations to the compiler, and then using a mix of compile-time validation and automatically injected code to make everything work transparently. The new(ish) annotation APIs make this process relatively easy.

Let's start with a task definition (Listing 1). A task is specified with the @Task annotation preceding the class definition. This annotation causes several things to happen. First of all, the annotation processor generates a subclass that does the following:

1. The subclass contains a dispatcher equivalent to the BlockingQueue and associated thread in the

earlier example.

2. The subclass constructor takes care of creating and starting that dispatcher thread. This thread dequeues requests and executes them.

3. The subclass provides overrides for every public method of the superclass. The overrides create the equivalent of the Runnable objects in the previous example. Each of these Runnables calls the equivalent superclass method and then handles return-value processing. The subclass overrides enqueue the Runnable objects on the dispatch queue.

In addition, the annotation preprocessor guarantees that certain conditions hold in the superclass (and prints an error message if the class isn't valid):

1. There are no protected fields. A protected field can be modified without synchronization by a subclass, so it isn't safe. protected methods are safe only because protected fields are not permitted.

2. There are no public fields that are not also static and if they're references, reference "read-only" objects. Since a public field can be modified by a thread other than the dispatch thread, it just isn't safe to expose it unless it's read-only. A read-only field either holds a primitive-type value, or it's a reference to an immutable object. By "immutable" I mean that all fields of the referenced object are either final instances of basic types or final references to classes. The rule is applied recursively to all referenced objects. String and Number objects are immutable, for example. If immutability can't be guaranteed at compile time (if the field is an interface reference, for example), then the generated subclass's constructor checks for immutability at run time.

3. There are no public final or static methods. A static method could be called by another thread, and could access other static fields that might be used by the dispatch thread. Public final fields are prohibited because Java's annotation mechanism doesn't let us modify existing classes—everything has to be done by overriding methods. You can't override a final method, though. That's why rules like this also exist in annotation-based frameworks like Mockito.

4. The Task class itself may not be final for the reason I just mentioned.

5. No Task can access the private or protected fields of another Task, even if they're instances of the same class. Think of private as object-level, not class-level, privacy.

6. A method may return a local variable, a synthesized value, or a read-only field, only. Code is generated in every subclass override that enforces this rule. If you need to return the value of a modifiable field, return a copy. This isn't a bomb-proof test because a badly written method could insert a non-read-only field into a third object and return that third object, but it's better than nothing. I may add code that recursively checks all the fields of all returned objects, but that's quite a bit of runtime overhead. I'd be interested in your comments on this one. It's okay to return "unmodifiable" versions of the standard Collection classes, as are generated by various Collections-class methods.

7. Method arguments must be primitive types or references to read-only objects. This rule is also enforced at runtime if it can't be verified at compile time. It's okay to pass "unmodifiable" versions of the standard Collection classes, though this exception is risky because it's possible to modify the wrapped collection after it's been wrapped.

8. Method arguments may be assigned to fields only if they're read-only. Code is generated in every subclass override to enforce this rule. If you need to preserve a non-read-only-argument's value

between calls, store a copy.

Essentially, all these rules are attempting to ensure that no objects used by the Task are modifiable by other threads. The rules aren't perfect—we'd really need to modify the language itself to handle every case—but they find the majority of problems.

In terms of programming, you don't need to worry about synchronization at all. You'll never use the synchronized keyword or the java.util.Concurrent classes. Just program as if the Task were single-threaded.

```
 1 @Task
 2 class MyTask    // may not extend anything
 3 {
 4     private method(){...};  // is okay
 5
 6     public int getX()
 7     {   return x;   // illegal
 8
 9     public void setX(int val)
10     {   x = val;    // illegal
11     }
12
13     public void f( Cls args            ) // okay if immutable
14     public void g( @Unmodifiable Cls2 arg ) // suppresses all
                    // checking for unmodifiability/immutability.
15                  // (required for some legacy classes).
16
17     Response<T> doSomething( ArgVal val )
18     {
19         T returnValue;
20         //...
21         return new Response<T>(returnValue);
22     }
23 }
```

Listing 1: A Task Definition

Methods that don't return void must return a Response<T>, a wrapper around the actual return value. Remember, all methods are effectively asynchronous, so the return value won't be valid until the method completes, which could be a considerable time after the called method has returned. Response<T>, is an interface that extends java.util.concurrent.Future<V>, so can be used exactly like a Future if you have an existing multithreaded system. Among other things, you can wait on the Future for the request to complete, and you can abort the request (which in this case means remove the request from the queue). The Response<T>, defines a few additional methods that I'll discuss in a moment.

Though all of the above seems like a lot of rules to remember, in practice it's not such a big deal. Just make

everything that you can private, and remember that nobody outside the class is allowed to access local fields, and that objects that come in from outside are also potentially dangerous, so make copies of these objects before storing them. The rules help you get things right by pointing out dangerous situations.

I should also say that I've deliberately not made things safe as a side effect, because that can violate the Least-Astonishment principle. For example, if you pass a method a standard Collection object, I don't wrap it in an unmodifiable variant automatically because the code within the method won't be expecting that change. I'd rather have a compile-time error than a mysterious runtime exception.

Talking to the Task

The notion of a BlockingQueue is handy when thinking about Active Object, but it turns out that it's much more useful to talk to tasks using a messaging system—a very lightweight and entirely in-memory system that works much like JMS.

Messaging systems support queues, of course, and that's the default way that things work in Tasks. A task effectively has a queue attached to it, and the methods of the task enqueue requests, then return. A "message," in this context, is an enqueued request.

However, messaging systems also support the notion of topics, and that's occasionally handy. The basic difference between a topic and a queue has to do with how many objects are involved in the communication. In the case of a queue, only one object—the Task—receives requests. In the case of a topic, several Tasks could "subscribe" to the topic and be notified when other tasks (or just plain objects) "publish" to the same topic. In JMS, the published request could be handled by one or all of the subscribers, but in the current task framework, the request is handled by a single Task. You don't know which one, however.

In the current Task implementation, queues are the default way of doing things, and the queues aren't named —the Task-object reference serves the same purpose. If you want to use topic-based communication, you need to add an argument to the Task annotation:

```
@Task(useTopics)
```

All methods of all instances publish to the topic—it doesn't matter which instance you use to publish the request. That is, no matter how many instances you create, there will be only a single dispatcher thread and a single queue. All Task instances will effectively publish to that single queue. All instances of all tasks queue up at the read end of the single queue, and the instance that handles the request may not be the one through which you published the request.

Work is distributed on a first-available basis to all the waiting tasks, which run in parallel. The other way you can think of it is that when topics are on the scene, all instances of the Task share a single dispatch queue, and publishing to a topic-based Task effectively queues the request up on that single queue. Requests are dispatched to whatever Task objects are waiting. The mechanism is similar to a thread pool.

```
@Task(useTopics=true, instances="3")     // instances can be any
                // positive number or 0 (the default if missing).
```

As a convenience, you can specify the number of Task instances in the annotation, and then you don't have to explicitly create any of them. They'll be created by a machine-generated static initializer, and you can send a message as if you were using a Singleton. That is, given

```
@Task(useTopics=true, instances="3")
public class MyTopicClass

{
    public void doSomething(){ /*...*/ }
}
```

you publish to the topic like this:

```
MyTopicClass.publish().doSomething();
```

Think of publish() as the equivalent of getInstance() on a Singleton.

Bear in mind that individual tasks cannot (easily) talk to one another. They can't access each other's private fields, for example, so topics are useful primarily for doing things that can be done independently: managing a pool of database connections, for example, or handling multiple, but independent, HTTP/REST requests sent to a web server. You'd have one database connection (or one HTTP client) per Task. The tasks run in parallel, but they are effectively independent of one another.

Getting Responses (Return Values)

So far, we've looked at sending requests to the Task, but how do you get a response? Since a Response is a Future, you can use the existing java.util.Concurrent mechanisms, but that's supported primarily to talk to legacy code. Using a Future is easy, though, because that's what the request actually returns:

```
Future<T> t = myTask.doSomething(val);
```

You can also wait for the task to complete like this:

```
Future<T> t = myTask.doSomething(val).thenWait();
```

Finally, when you want a response to be handled by the task that sends the request, use:

```
receivingTask.doSomething(val).thenDo(this, (v)->{ ... f(v); ... } );
```

One task sends a request to another task (the receivingTask, above). When that task is done, it sends another request object (specified in the lambda) back to the originating task (this). That is, the lambda is wrapped up as if it were a standard request and then queued up on the request queue of the task specified in the first argument to thenDo() (typically this). That task will run the lambda when it gets around to it, just like any other asynchronous call, so everything remains single-threaded (and thread safe).

It's important to realize that the code in the lambda is executed in the context of the sending, not the receiving, Task. An exception is thrown if the first argument to thenDo() is a topic-style Task or a normal class, declared without the @Task annotation. That is, this mechanism is meant to send a response to the Task that made the original request only.

Finally, the call:

```
receivingTask.doSomething(val).thenExecute( (v)->{ ... f(v); ... } );
```

executes the specified lambda when the request completes. This method is for non-Task objects that are delegating to Tasks. The lambda is executed on a pooled thread. Since the lambda can access local fields in the calling class, there are potential synchronization issues here, so be very careful.

Conclusion

So that's the core of the Active Object framework. The main reason for doing all this work is to make it easy to program asynchronous tasks. The code within the Task, since it's effectively single-threaded, doesn't require any synchronization at all, so is easy to write, debug, and maintain. If the entire program is task-based, then that same ease of maintenance applies to the entire system. The one problem the framework can't solve is synchronization when you mix Task objects with standard multithreaded code, but you can't do everything.

Just to reiterate what I said at the beginning of this article: the framework is partially built at present, and is definitely a work in progress. The main reason for writing this article is to get some feedback before I spend any time chasing down the wrong path, so if you have comments, please send them to jao@holub.com. I'll summarize the comments and post them to a JAO page on my website. Eventually, the entire framework will be published as open source, and there will be a book describing how it works and how to use it in much more depth than I've done here, but you've got to start somewhere!

Allen's long-running columns for Dr. Dobb's Journal (C Chest), JavaWorld (Java Toolbox), SD Times (JavaWatch), and others have been highly influential in the industry. His open-source Java threading library predated the java.util.concurrent system, and the associated book (Taming Java Threads) was one of Apress's all-time best sellers. He's written a total of nine books on programming topics. More info at his site.

Send the author your feedback or discuss the article in the magazine forum.

Search Magazine TOC

Go

- All magazines
- Discuss the magazine

You may also like…

Functional Programming in Java Venkat Subramaniam
