JOURNAL OF OBJECT TECHNOLOGY

Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 7, July-August 2004

The Imperion Threading System

Douglas Lyon, Fairfield University, Fairfield

When you come to
a fork in the road...
Take it.

- Yogi Berra

Abstract

This paper describes the use of the command, facade and decorator design patterns to alter the interface and add new responsibilities to threads. Termed *Project Imperion*, the threads are developed with the same design patterns as previously reported for Imperion GUI components [Lyon 2004b]. The benefits of Imperion threading include: simplifying code, easing maintenance, separation of thread management logic from the business logic and improved reliability.

Our experience shows that threads are often run more than once. They are frequently stopped and then restarted. However, the present mechanisms for doing this are both low-level and exception-prone. Imperion threads have built-in support for iteration. They support killing and restarting threads by introducing a new class called the *RunJob*. The *RunJob* tracks the number of times it has been run, and can be set to run only so many times before it dies. It can be set to start automatically, or be set to wait until explicitly started (or restarted).

The Imperion threading system is a more reliable threading system than the normal *java.lang.Thread*. Imperion removes dangerous methods and guards' inputs in order to avoid exceptions. For example, the daemon property can only be set during construction, priorities are guarded for correctness and restarting is safe. This avoids a fruitful source of complex run-time errors that have been bugging both novice and seasoned programmers since the release of JDK 1.0.

Project Imperion was named for the Latin root, *imperium*, which means the power to command. Like the Intel use of the word CELERON, the *on* suffix was added to give the word a high-tech look (like electron, proton or muon). It was first conceived at the skunk works of DocJava, Inc., in the late 1990's.

1 THE CURRENT THREAD MODEL

There are two ways to make a new thread. Either by subclassing the *Thread* class, or by passing an instance of a class that implements the *Runnable* interface as a parameter to a

Cite as follows: Douglas Lyon: "The Imperion Threading System", in *Journal of Object Technology*, vol. 3, no. 7, July-August 2004, pp. 57-70.

thread class constructor. Invoking the *start()* method causes a thread to move into the *ready* state. Threads in the *ready* state are queued for execution but are not running.

Threads have a property, called the *Daemon*. The term daemon (old English spelling of *demon*) is a term in the operating systems field. Some have said that it stands for "Disk And Execution MONitor". A daemon is a task or a thread that remains idle until an event occurs. Daemon tasks are often left idle on an operating system. For example, MacOS X typically has 41 daemon tasks sitting idle. In comparison, it is not uncommon for RedHat Linux to have over 100 idle tasks.

Java can start threads or tasks. To start tasks, it is typical to use platform dependent (i.e., non-portable) code. In Java a daemon thread has special meaning. This meaning comes from the Sun API. In Java, the daemon thread dies when all non-daemon threads die. Before it dies, a daemon is said to be *lurking*. To put it another way, when only daemons remain in a program, the program exits. For example, a print spool daemon will wake when it sees a file in its spool directory, it then wakes up and prints the file. The print daemon goes back to sleep after all the files in its spool directory are printed. Typically Unix systems run daemon tasks in order to handle requests for services. They are typically started by single task (called *inetd*). Examples of elements started by inetd (or on RedHat Linux, /etc/xinetd.d) include, *echo*, a daemon that responds to a *ping*. There are day daemons, time daemons, login daemons, printing daemons, garbage collector daemons, etc.

Any thread may be set to be a daemon thread by using the *setDaemon(true)* invocation. However, this can cause an *IllegalStateException* if the thread is currently alive. Thus, the time-window to safely set the daemon property lies between the construction of and the starting of a thread. The following example prints out a threads' string representation, its name, and its daemon state:

```
1. class TestThread extends Thread {

2.      public void run() {
3.              while (true) {
4.                      System.out.println("Priority=\t" +
                getPriority());
5.                      System.out.println("toString=\t"+toString());
6.                      System.out.println("getName=\t"+ getName());
7.                      System.out.println("isDaemon=\t"+isDaemon());
8.                      System.out.println("isAlive=\t"+isAlive());
9.                      try {Thread.sleep(10000);}
10.                     catch (InterruptedException e) {}
11.                     }
12.     }
}
```

To run an instance of the *TestThread*, you must make an instance of the *TestThread* and start it.

```
TestThread tt = new TestThread();
tt.start();
```

The following will be printed every 10 seconds at the console:

```
Priority=5
toString=Thread[Thread-2,5,main]
getName= Thread-2
isDaemon=false
isAlive= true
```

Line 9 puts the thread to sleep for 10,000 milliseconds (10 seconds). *Thread.sleep* takes a long integer because 32 bits does not have enough range to represent long time periods. For example, there are $2^{\log_2(1000*60*60*24)} = 2^{26}$ milliseconds in a day. A signed 32-bit integer overflows in $2^{31}-1$ milliseconds, (3.5 weeks). A long, 64-bit integer will overflow in 292.4 billion years. The Suns' corona will have engulfed the Earth in only 10 billion years (by which time, even online journals will be out of print) [Lyon 1999]. Working in milliseconds is counter-intuitive for most people and ignores the nanosecond resolution available in modern high performance operating systems and CPUs. To address this concern, a double precision float is use to represent the time, in seconds, for Imperion threads. This is translated into the *Thread.sleep* method, which still takes time in milliseconds, at present. There is already a proposal on the table for high-resolution clocks for real-time threads, available at http://www.rtsj.org/. Even so, the present use of a double precision number to represent time does not add any precision to the sleep method, at the moment. The *Thread.sleep* method can throw an *InterruptedException* if the thread is interrupted while it is sleeping. Otherwise it will exit normally.

Killing a thread is problematic. The *stop*, *resume* and *suspend* methods have become deprecated because they are considered unsafe. Suspend is inherently deadlock-prone and stopping a thread can cause instance corruption by releasing all the locks on the resources needed by the thread operation. Since the state of the resources cannot be predicted, program execution can't reliably proceed. The run-time errors that may result can manifest themselves at any time in the future. To more deeply understand why this is true, we need to examine the life cycle of a thread, as shown in Figure 1.

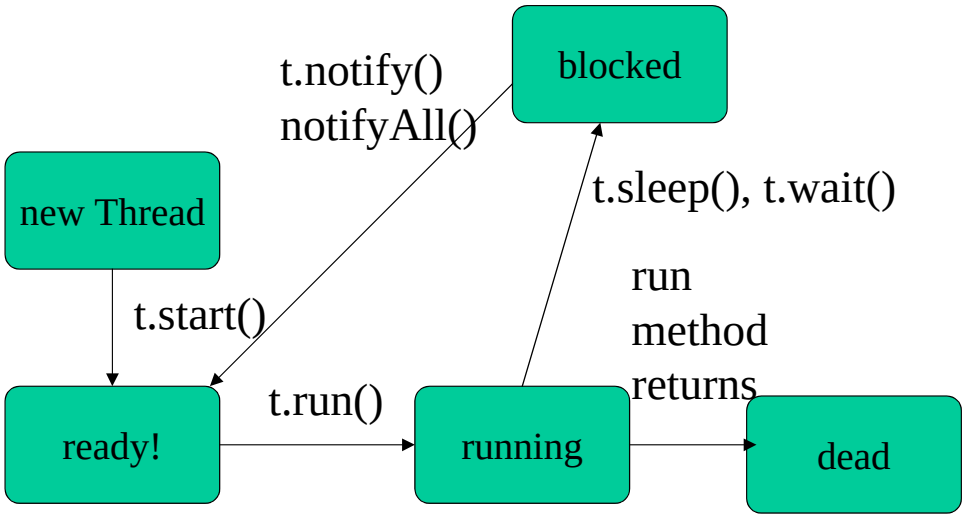


Figure 1. The Life Cycle of a Thread

Figure 1 shows that when the *run* method returns, the thread becomes dead. Thus, if we kill a thread while it is running, we cannot be assured that it is finished with whatever it is doing. If the thread is blocked when we kill it, it may be waiting for a resource to become available, and it will continue processing when it wakes up. For example, suppose my interest computation is held up because someone is making a transaction with an ATM. If I interrupt the interest computation with an asynchronous stop invocation, then the interest could be lost.

To see how deeply the thread life cycle is embedded in the language, we need look no further than the implementation of the thread life-cycle methods. A blocked thread can be restarted by invoking the *notify* method. The *notify* method is defined in the *Object* class. The *notify* method is used to notify an object of a change in condition. The methods *wait*, *notify* and *notifyAll* are methods defined in the *Object* class. The *wait* method causes the thread of execution to block execution until its *notify* method is invoked by another thread. The *wait* method can take three forms:

```
wait();
wait(long milliseconds);
wait(long milliseconds, int nanoseconds);
```

To more deeply understand why it is so important to allow threads to die only when their run methods end, we must see how race conditions between threads can occur. Race conditions occur when two or more threads try to access the same memory at the same time. In order to make code thread-safe, we have to ensure that certain operations happen all at once (these are known as atomic operations). To create atomic operations, we use the *synchronized* keyword.

The *synchronized* keyword may be applied to any reference data-type. For example, suppose there are several threads that are trying to perform an output operation to the *PrintStream* instance contained in the *System* class (i.e., *System.out*). If the threads all output to *System.out* asynchronously with respect to one-another, then they will tend to over-write one another, intermixing each-others' output. The solution is to place a *lock* on the *System.out* resource using *synchronized*. The code will have the form:

```
synchronized(System.out) {
   System.out.println(...);
...
}
```

As another example, suppose that we are programming a banking system. When someone adds money to the bank, we execute:

```
public void addMoney(BankAccount ba, double dollars) {
   System.out.println("this is going to lock the account");
   synchronized(ba) {
         ba.addMoney(dollars);
   }
   System.out.println("the lock is released");
}
```

The lock implements a form of *mutex* (mutual exclusion). Should the resource that is locked become unavailable (or blocked) it then becomes a single-point of failure. Imagine

if such a single point of failure existed in a mission-critical sub-system (like the communications port between the tail-section and cockpit of an air frame). All threads that depended on the correct operation of the port would become *deadlocked*. The deadlock condition occurs when two or more threads are unable to make progress, due to a dependency on an unavailable resource. This could easily happen if a thread is suspended in the middle of executing on a synchronized resource. If deadlock occurs in a mission critical system, it might result in a mission failure.

To implement atomic execution on a block of code, consider making the entire method *synchronized*. For example:

```
1. class Animation implements Runnable {
2. public void synchronized run() {
3. ....
4.        try {
5.              Thread.wait();
6.        }
7.        catch (InterruptedException e) {
8.              ....
9.        }
10.}
11.}
```

Line number 5 is used to give other threads a chance to run. During the execution of the *run* method, no other threads may execute.

It is also possible to synchronize on the current instance. For example:

```
synchronized(this) {
   .....
}
```

Synchronizing on the current instance is considered less clear than synchronizing at the method level [Campione and Walrath].

2 IMPERION THREADS

This section presents an approach to running threads by creating a new, more feature-rich and more reliable thread container, called the *RunJob*. A *RunJob* instance contains an instance of a thread. The *RunJob* contains enough information to start and stop the thread, and to have the thread run any number of times, with a given delay between executions. A *RunJob* has an implementation of a run method and so knows how to run itself. Thus making use of the *command design pattern*. The command design pattern places an instance of a command into an instance of another class and calls the *issuer*. For example, a button can have the role of the issuer, holding a reference to an instance of a command.

The *RunJob* uses the façade design pattern since it provides a simpler interface to the *Thread* class. It also uses the decorator design pattern in that it is adding responsibilities to the *Thread* class. Since the *RunJob* makes use of both design patterns, we have named it the *decorator-façade* pattern.

The Imperion *RunJob* has an overloaded constructor. The most general one follows:

```
public RunJob(double seconds,
              boolean wait,
              int count,
              boolean isDaemon)
```

An instance of the *RunJob* runs every so many *seconds*. If *wait* is *true* then the *RunJob* will not start unless the start method is invoked, otherwise the *RunJob* starts right away. If the count is present, then the job only executes *count* times. The *isDaemon* method marks this *RunJob* as either a daemon *RunJob* or a user *RunJob*. The Java Virtual Machine exits when the only *RunJobs* running are all daemon *RunJobs* and all the threads are daemon threads. Stopping the job yields CPU resources to other threads. The *isDaemon* property can only be set at construction time and is immutable during the life of the *RunJob*. This avoids a source of run-time exceptions.

We define the *RunJob* class as one that contains a *CommandThread instance*. It works by implementing the *Runnable* interface in a callback mechanism and invokes the *run* method in the *RunJob* at given intervals. The *run* method is left undefined and this causes the *RunJob* class to be *abstract*. The following example prints the date and time every two seconds:

```
public static void main(String args[]) {
    // Anonymous inner class
    // That uses the command pattern
    // It also uses decorator-facade pattern
    new RunJob(2) {
        public void run() {
            System.out.println(
                    new java.util.Date());
        }
    };
}
```

The following example will wait for the start method to be invoked before starting the *RunJob*. It will then run every 1.5 seconds:

```
RunJob rj = new RunJob(1.5,true) {
            public void run() {
                System.out.println(
                        new java.util.Date());
            }
        };
        rj.start();
```

The following example starts the *RunJob* right away, then stops and restarts the job. It will only run 5 times before the job dies, even if the job is restarted. Thus, the *RunJob* instance tracks the number of times it has been run.

```
RunJob rj = new RunJob(1.5, false, 5) {
            public void run() {
                System.out.println(
                        new java.util.Date());
            }
        };
        rj.stop();
        rj.start();
    }
```

The astute reader will notice that anonymous inner classes are used to define the *RunJob*. This is deliberate. Anonymous inner class bodies are generally kept short, to improve readability (and to help to separate the thread logic from the business logic). Also, the anonymous *RunJob* classes promote the isomorphic mapping between instances of the *RunJob* and instances of a thread. Having one instance of a class that implements runnable for each instance of a thread is a well-known strategy for creating threads that have their own instance variable. In comparison, having a single runnable instance passed to multiple threads can allow all the threads to freely manipulate variables in the runnable instances, leading to potential inconsistencies [Sandén 2004].

To better understand the rationale for the Imperion threading system, it is necessary to understand the history of the thread and some of the shortcomings that it has. In the standard method of using a thread a deadlock-prone method for suspending and resuming a thread can easily be implemented with code in the following format:

```
private volatile Thread t;
private boolean threadSuspended;

    public void doProcess() {
        if (threadSuspended)
            t.resume();
        else
            t.suspend();  // DEADLOCK-PRONE!
        threadSuspended = !threadSuspended;
    }
```

Thus, it was typical to suspend and restart threads before the *resume* and *suspend* methods became deprecated. The Sun suggested alternative is to use code of the form:

```
public synchronized void doProcess() {
        threadSuspended = !threadSuspended;
        if (!threadSuspended)
            notify();
    }
```

It is up to the programmer to add the following code to the "run loop":

```
                synchronized(this) {
                    while (threadSuspended)
                        wait();
                }
```

I take issue with this approach. First, the programmer has to remember to do something (which means that there will be occasions when the programmer forgets to do something). Second, the rationale may be clear to the author of the code, but not to the maintainer of the code (who may remove the synchronized invocation and find that no

harm was done, *this time*). Finally, there is an additional cost of synchronization that is imposed by such an implementation. The synchronization is required in order to avoid race conditions.

The *RunJob*, as implemented in Imperion, treats a *stop* invocation as non-urgent. The containing thread is always allowed to finish its present command before a flag is checked and the thread stops, leaving instances in known states. Further, the start method makes a new thread instance, reusing the old implementation for the thread command, but resetting the state to the old state (i.e., it tracks the number of times the old thread was run, stores its priority, name, daemon property, etc). As a result we are encumbered with the overhead of making a new instance and setting it to the old thread member variable. This trades the overhead of synchronization for the overhead of creating a new thread. The question of which takes longer really depends on how much work is being done in the synchronized method. The overhead for making a new instance of a thread should at least be a fixed one.

3 WHAT'S THE ASSOCIATION BETWEEN THE *RUNJOB* AND THE *THREAD*?

There is a composition association between the *RunJob* and the *Thread* classes. That is, the *RunJob* uses a *Thread* to run the contained command.

Using the *Facade* design pattern we are able to map several common components to take advantage of the *Runnable* interface, as shown in Figure 1.

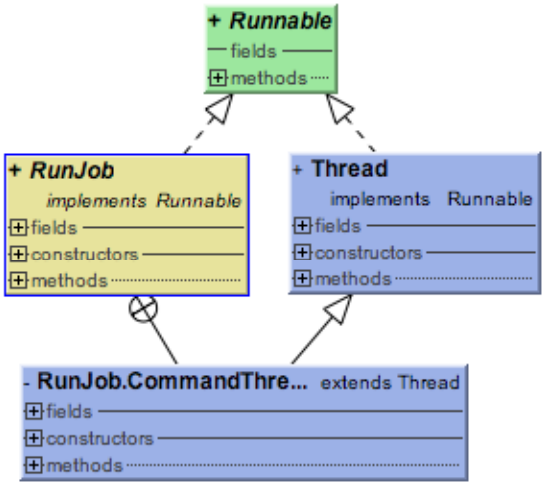
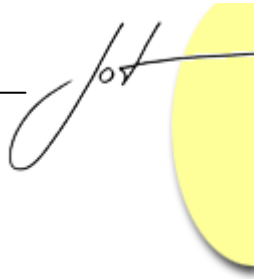


Figure 2. The *RunJob* Associations

At the heart of the *RunJob* is the inner*CommandThread* class that invokes the call-back *run* method in the *Job* class:



```
private class CommandThread extends Thread {
        RunJob job = null;


        CommandThread(RunJob job) {
            this.job = job;
        }

        private boolean isCountDone() {
            if (count == Integer.MIN_VALUE)
                return false;
            return numberOfTimesRun > count;
        }

        public void run() {
            while (cowsComeHome) {
                numberOfTimesRun++;
                if (isCountDone()) return;
                job.run();
                try {
                    Thread.sleep(ms);
                } catch (InterruptedException e) {
                }
            }
        }
    }
```

Invoking a *wait* (or *sleep*) method inside of a synchronized block is likely to result in a deadlock that can come from holding the lock for an indefinite period of time on some resource. The synchronized blocks of an Imperion thread will always be inside of the primary iteration block, whose end task is to run a non-synchronized sleep. This should help to avoid some deadlocks.

4 WHY USE COMPOSITION RATHER THAN EXTENSION?

Several of the methods in the *Thread* class can throw exceptions if they are misused. Overriding these methods is not an option, as several are declared as *final*. Further, the *RunJob* adds new responsibilities that the *Thread* class was never designed for (i.e., being restartable, being able to track the number of times the *run* method is invoked, etc.).

More insidious is the need to materially alter the interface of the *Thread* class in order to hide some of the methods. It is a syntax error for a sub-class to make the visibility of shadowed methods more restrictive. Finally, there is no good way to un-deprecate a method, like the *stop* method in the Thread class, except by re-implementing it in the containing class.

Threading logic is often added on an as-needed basis. In fact, the *RunJob* is really just a collection of handy thread features that enable the command and control of a thread. Is adding another class for wrappering a thread invocation really needed?

In my view there are compelling reasons to hide the complexity of thread logic needed to stop and start threads while maintaining some semblance of state, and to make the thread logic as easy as possible for the programmer to maintain. For instance, the following *RunColorButton* class automatically changes its background color in order to alert the user of its importance:

```
public abstract class RunColorButton
        extends RunButton {
    private RunJob j = new RunJob(1, true) {
        public void run() {
            setBackground(
                    MathUtils.getRandomColor());
        }
    };

    public RunColorButton(String s) {
        super(s);
        j.start();
    }
    private boolean running = true;
    public void toggle() {
        if (running){
            running = false;
            stop();
            return;
        }
        running = true;
        start();
    }

    public void stop() {
        j.stop();
    }

    public void start() {
        j.start();
    }
}
```

The above code automatically handles the starting and stopping of the background color change, by reusing the logic in the *RunJob* class. If this complexity were not hidden from view, the code would more than double in size. Even worse, it would contain code that was practically duplicated in other classes.

Figure 3 The flashing color panel

The code for implementing the color panel follows:

```
public class ColorPanel extends JPanel {
    public static void main(String args[]) {
        class ColorFrame extends ClosableJFrame {
            ColorFrame() {
                super("Colors!");
                Container c = getContentPane();
                c.setLayout(new FlowLayout());
                c.add(new ColorPanel());
                setSize(200, 200);
                show();

            }
        }
        new ColorFrame();


    }

    RunJob job = new RunJob(1, true) {
        public void run() {
            Color randomColor = MathUtils.getRandomColor();
            setBackground(randomColor);
            setForeground(randomColor);
        }
    };

    ColorPanel() {
        super();
        setLayout(new FlowLayout());
        add(new RunColorButton("[ggo") {
            public void run() {
                job.start();
                toggle();
            }
        });
        add(new RunColorButton("[wstop") {
            public void run() {
                job.stop();
                toggle();
            }
        });
    }
}
```

The color panel shows that the buttons not only stop and start their own colors from changing, but that of the containing panel as well. The *RunJob* simplifies the use of threading and its ease of use actually encourages the use of threading.

5 CONCLUSION

The use of the command and facade design patterns to simplify thread usage is not new [Lyon 2004]. However, the combination of the facade and command patterns, along with the goals of improved reliability and ease of use are new. The danger of run-time exceptions is greatly reduced with the Imperion threading system. This is due, in part, to the guarding of the input. For example, it causes a run-time exception to alter the daemon parameter after starting a thread. As a result, the daemon parameter can only be set in the constructor of a *RunJob*, thus eliminating a source of a run-time exception. Also guarded is the input to the priority setting method. *RunJob* thread properties can fall out of sync with one another if the security manager blocks the setting of a property.

A grouping of a thread logic framework into the *RunJob* factory methods improves maintainability and readability. The use of the anonymous inner classes promotes short method bodies (since people don't generally like to see long anonymous inner classes). The short method bodies, in turn, promote method forwarding to business logic (improving code reuse and promoting the separation of thread code and business logic).

One of the drawbacks of the Imperion system is that the *RunJob*s map their actions to only a single listener (themselves). This would seem, on the surface, to be a big limitation. For example, what if many instances are interested in the thread's execution? I suggest that if this occurs, a new class is needed with a new responsibility that involves the observer-observable design pattern.

Having instances listen to their own event is an easy limitation to live with. In fact, I contend that this is preferable, as it limits inter-object associations, which is, in my view, a primary metric of object-oriented complexity.

The Imperion project is an open-source project freely available at http://www.docjava.com.

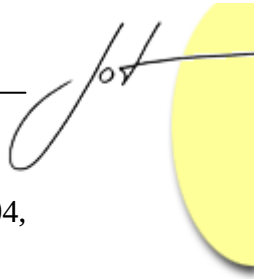
REFERENCES

[Camp1996] Campione and Walrath, *The Java Tutorial*, Addison Wesley, 1996.

[Lyon 1997] Douglas A. Lyon and H. Rao, *Java Digital Signal Processing*, M&T Press, 1997. Available from http://www.docjava.com.

[Lyon 2004] Douglas A. Lyon, *Java for Programmers*, Prentice Hall, 2004. Available from http://www.docjava.com.

[Lyon 2004b] Douglas A. Lyon, "Project Imperion: New Semantics, Facade and Command Design Patterns for Swing", in *Journal of Object Technology*, Publication Pending.



[Sand 2004] Bo Sandén, “Coping with Java Threads”, in *IEEE Computer*, April 2004, V. 37, N. 4, pp. 20-27

About the author

After receiving his Ph.D. from Rensselaer Polytechnic Institute, **Dr. Lyon** worked at AT&T Bell Laboratories. He has also worked for the Jet Propulsion Laboratory at the California Institute of Technology. He is currently the Chairman of the Computer Engineering Department at Fairfield University, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. E-mail Dr. Lyon at Lyon@DocJava.com. His website is http://www.DocJava.com.

Appendix A. – The Java Doc

What follows is the javadoc for the *RunJob*:

public abstract class **gui.run.RunJob** implements java.lang.Runnable

Constructors

public RunJob(double seconds)
public RunJob(double seconds, boolean wait)
public RunJob(double seconds, boolean wait, int count)
public RunJob(double seconds, boolean wait, int count, boolean isDaemon)

Run the *RunJob* every *seconds.* If wait is *true* then do not start the *RunJob* until the start method is invoked. Otherwise, start right away. If the *count* is present, then the job only executes *count times*. IsDaemon marks this *RunJob* as either a daemon *RunJob* or a user *RunJob*. The Java Virtual Machine exits when the only *RunJob* running are all daemon *RunJobs* and all Threads are daemon threads.

Methods

public int getNumberOfTimesRun() - The number of times the *RunJob* was run.

public void setPriority(int priority) - Changes the priority of this *RunJob*. This may result in throwing a *SecurityException*. Otherwise, the priority of this *RunJob* is set to the smaller of the specified new Priority and the maximum permitted priority of the *RunJob's* thread group. If the priority is smaller than MIN_PRIORITY then it will be set to MIN_PRIORITY. If the priority is greater than MAX_PRIORITY then it will be set to MAX_PRIORITY. This avoids illegal argument exceptions at run-time.

Parameters priority - priority to set this *RunJob* to

Throws SecurityException - if the current *RunJob* cannot modify its thread.

public final int getPriority() - Returns this *RunJob's* priority.

public void start() - Start are restart a *RunJob* that was stopped.

public void setName(String name) - Changes the name of this *RunJob* (and thread upon which the RunJob is based, to be equal to the argument name. First the *checkAccess* method of this thread is called with no arguments. This may result in throwing a *SecurityException*.

public java.lang.String getName() - Gets the name of this *RunJob* (and the thread upon which the *RunJob* is based First the *checkAccess* method of this thread is called with no arguments. This may result in throwing a *SecurityException*.

public void stop() - Stops the *RunJob* by causing a thread death. A start() invocation Will make a new thread and restart the *RunJob*.

public boolean isDaemon() - Tests if the *RunJob* is a daemon *RunJob*.

public static void main(String[] args) – A test method.

public boolean isAlive() - Tests if this *RunJob* is alive. A *RunJob* is alive if it has been started and has not yet run for the requisite number of times.

public java.lang.String toString() - Returns a string representation of this *RunJob*, including the *RunJob's* name, priority, and thread group to which the *RunJob's* thread belongs.

Fields

public static final MIN_PRIORITY - The minimum priority that a *RunJob* can have.

public static final NORM_PRIORITY - The default priority that is assigned to a *RunJob*.

public static final MAX_PRIORITY - The maximum priority that a *RunJob* can have.