Thread Synchronization Policies in DrJava

Since DrJava is built using the Java Swing library, it must conform to the synchronization policies for Swing. Unfortunately, the official Swing documentation is sparse and misleading in places, so this document includes a discussion of the Swing synchronization policies.

The Architecture of DrJava DrJava is a pure Java application involving two Java Virtual Machines (JVMs): (i) a master JVM that supports the user interface, the DrJava editor, and DrJava compilation; and (ii) a slave JVM that runs the interpreter and unit tests. DrJava currently uses the Java RMI library to support communication between the master and slave JVMs. In the future, a lighter weight communication mechanism may be used instead of RMI.

Every Java Virtual Machine (JVM) includes several independent *threads* of execution. At a minimum, a JVM includes:

- a *main* thead that begins program execution with the main method in the program's root class;
- an *event-handling* thread (henceforth called the *event* thread for short) that process all input events including keystrokes and GUI actions such as moving the mouse and depressing buttons;
- a *garbage-collection* thread that performs garbage collection either intermittently or continuously; and
- a *finalization* thread that executes finalize methods for objects after they become unreachable.

The behavior of the garbage-collection and finalization threads does not affect the behavior of the rest of the program (ignoring performance) with one critical exception. In objects that include a finalize method, this method can be executed exactly *once* at any time after the object becomes unreachable. Since there is no guarantee that finalize methods will executed in a timely fashion, their usage in DrJava is confined to collecting information during program testing. For this reason, we can safely ignore both the garbage-collection and finalization threads when thinking about thread synchronization in DrJava.

Synchronization in Swing For the sake of simplicity and reduced overhead, most Swing classes do not perform any explicit synchronization (locking) operations. The general rule is that all Swing code must be executed in the event thread. When this rule is followed, no synchronization problems involving Swing operations can arise because all of the code for these operations is executed sequentially in the same thread.

The event thread is simply an endless loop that removes the next event from the Java event queue and processes it, executing any listeners that have registered for the event. No actions are performed by the event thread outside of this strict "first-come-first-served" order processing of GUI events. In simple Swing applications, all program code (except for some initialization code) runs in the event thread; the main thread terminates (dies) after the initialization phase just as the event thread is effectively activated.[1]

In more complex Swing applications like DrJava, this simple discipline does not suffice because some GUI events take a long time to process. During the processing of such an event, the processing of all other events is inhibited (locked out). As a result, the application cannot respond to any other event requests (no matter how trivial) in the interim. This limitation can be overcome by spawning new threads to process events that may take a long time to process. Unfortunately, the logical complexity introduced by spawning new threads is *enormous*. Most apparently reliable multi-threaded programs are riddled with hidden synchronization bugs that are masked by the fact that the particular schedules required to reveal the bugs are not generated during program testing. At this stage in the evolution of DrJava, it is no exception. We have a long way to go before we can claim that DrJava is free or nearly free of synchronization bugs.

Complying with the Swing event-thread restriction The Swing library provides two static methods for remotely running code in the event thread:

```
void invokeLater(Runnable r)
void invokeAndWait(Runnable r)
```

These static methods are not very well-documented in the official Java documentation but they are found in *two* different places in the Java GUI libraries: the classes javax.swing.SwingUtilities and java.awt.EventQueue. The versions in java.awt.EventQueue may be slightly preferable because the versions in SwingUtilities simply forward their calls to the corresponding methods in EventQueue.

The method invokeLater(Runnable r) takes a Runnable (the class used to represent the *command* design pattern in the Java libraries) and places that action at the end of the Java event queue. The action is *asynchronous*: the method returns as soon as the specified action has been queued. The action is

---

[1] The details of this handoff are delicate. Some simple Swing applications occasionally fail because the handoff is not handled properly. Roughly speaking, a GUI component should not be touched except from the event thead once it has been realized, which happens when setVisible(true), pack(), show(), or validate() are invoked on the component. This rule is rough because it is often not clear when realizing one component indirectly realizes another component. It is perhaps better practice not to touch any GUI components from threads other than the event thread once any component has been realized. Note that some swing operations may not work reliably on GUI components that have not yet been realized. For example, the field of a JTextArea should not be set until it has been realized.

executed sometime in the indefinite future by the event thread after all previously queued actions have been performed. Hence, if some code following a call on invokeLater depends on the execution of the requested action, we have a serious synchronization problem. How do we know when the requested action has been executed?

The method invokeAndWait(Runnable r) is intended to address this problem. It takes a Runnable, places that action at the end of the Java event queue, and waits for the event thread to execute the action. But invokeAndWait has an annoying feature: it throws an exception when it is called from within the event thread. In a complex application like DrJava, the same method can be called from the event thread and other threads. In general, it is very difficult to determine which threads call which methods.

In DrJava, we have worked around the limitations and inconveniences of the Java invokeLater/invokeAndWait methods in the Java libraries by defining our own versions of these methods in the class edu.rice.cs.util.swing.Utilities. Our versions first test to see if the executing thread is the event thread. If the answer is affirmative, then our methods immediately execute the run() method passed in the Runnable argument and return. Otherwise, our invokeXXX methods call the corresponding method in java.awt.EventQueue. Note that our versions of the invokeXXX methods are very efficient if they happen to be called in the event thread because no context switching is required to perform the requested action.

In some situations, the distinction between the DrJava versions and the Sun versions of these primitives is critically important. In particular, it may be essential to run some code in the event thread *after* all of the pending events in the event queue have been processed. In these situations, DrJava must use the Sun versions of these methods. On the other hand, if a call on invokeAndWait can be executed in the event thread as well as other threads, DrJava must use our version of invokeAndWait. Similarly, if DrJava must run some Swing code asynchronously in an arbitrary thread after all of the events already in the event queue have been processed (for example, the execution of some notified listeners that run Swing code in the event thread using invokeLater) then DrJava must use the Sun version of invokeLater. In the absence of this timing constraint, the DrJava version of invokeLater is preferable to the Sun version because it performs the requested action more quickly if the event thread is executed.

The DrJava Utilities class also includes the method void clearEventQueue() which, when executed in a thread other than the event thread, forces all of the events currently in the event queue to be processed before proceeding. If it is executed in the event thread, it immediately returns because the event thread cannot wait on the completion of processing a pending event!

Exceptions to the Swing event-thread restriction The Swing event thread restriction is not tenable for some Swing data structures, particularly documents that contain editable text. Some Swing classes are intended to accessed by threads other than the event thread but this fact is not very

well-documented in Swing. Most Swing components have both a *view* (the GUI widget displayed on the screen) and a *model*, which is the data structure associated with or depicted by the view. For example, a Swing JTextPane has an associated Document that is displayed in the pane. While Document is an interface with unspecified synchronization policies, all of the implementations of the Document interface in Swing are derived from the abstract class AbstractDocument which includes a readers/writers synchronization protocol. To our knowledge, all of the public methods in these classes are thread-safe, even though most of them are not documented as such. (We believe that they must be thread-safe because the would break the execution of the methods that are documented as thread-safe if they weren't!)

All of the DrJava classes derived from AbstractDocument (including SwingDocument, AbstractDJDocument, DefinitionsDocument) conform to the readers/writers synchronization protocol established by AbstractDocument. In particular, any code segment in these classes that modifies instance fields must be bracketed by calls on writeLock() and writeUnlock(). Similarly, any code segment that only reads instance fields in these classes must be enclosed by calls on readLock() and readUnlock(). Note that the extent each such code segment is determined by whatever actions must be performed atomically with regard to the state of the instance fields in that class. For example, a code segment that appends text to the end of a document must perform writeLock() before reading the length of the document (using getLength()) and perform writeUnlock() after inserting the appended text (using insertText(…) at offset getLength().

The same readers/writers protocol is used in several DrJava classes that *decorate* classes derived from AbstractDocument. These classes include all classes that implement the DrJava ReadersWritersLocking interface such as ConsoleDocument and InteractionsDocument. Unfortunately, the writeLock() and writeUnlock() methods in AbstractDocument are not public, so they are renamed as acquireWriteLock() and releaseWriteLock() in ReadersWritersLocking because some (in fact most) classes that implement ReadersWritersLocking are derived from AbstractDocument. In in ReadersWritersLocking, the names of the readLock() and readUnlock() methods from AbstractDocument are similarly renamed as acquireReadLock() and releaseWriteLock()for the sake of naming consistency.

In contrast to JTextPane, the JList and JTree GUI widgets have associated models (DefaultListModel, DefaultTreeModel) that can *only be accessed in the event thread.* From the perspective of DrJava, this design choice is regrettable because it makes it more expensive and tedious to access and modify the models associated with these widgets. According the official documentation, there is no alternative to executing code that accesses DefaultListModel and DefaultTreeModel in the event thread. But we conjecture that the context switches involved in this approach would significantly degrade the responsiveness of DrJava. Consequently, we use a different synchronization policy that appears to be compatible with the assumptions made in the Swing library regarding these classes. In DrJava, we do not use the JList and JTree classes directly. We extend these classes by the DrJava classes JListNavigator/JListSortNavigator and JTreeSortNavigator. Our synchronization policy for these classes is based on the observation that all operations that change the state of the models in

ListNavigator/JListSortNavigator and JTreeSortNavigator are methods within these classes--with the exception of selecting the currently active item in the JListNavigator/JListSortNavigator or JTreeSortNavigator. This selection can be performed by triggering events corresponding to the JList and JTree GUI interface, *e.g.*, changing a selection using the mouse. Fortunately, the JList and JTree classes provide a hook for performing a client-specified action in the event thread when such a selection is made.

Since the current selection can be changed directly by GUI code, our solution is to keep a shadow copy of the current selection (called _current) in JListNavigator/JListSortNavigator and JTreeSortNavigator. The state of the shadow copy is updated using the hook method provided by the JList and JTree classes. Within DrJava the current selection is always read from the shadow copy. Note that the only Java code that can see an inconsistency between the current selection recorded in the Swing model and our shadow copy are document listeners that execute before the listener that updates the shadow copy.

Code segments in JListNavigator/JListSortNavigator and JTreeSortNavigator that modify the state of the associated model must run in the event thread. Hence, they must be embedded as commands within calls on the DrJava invokeXXX methods. These code segments must also synchronize on the associated model (the field named _model). Similarly, code segments in JListNavigator/JListSortNavigator and JTreeSortNavigator that access (read) the state of the associated model must synchronize on the model or run in the event thread. The explicit synchronization on the associated model enables code segments that only read the state of this model to run outside the event thread provided that they perform a read lock on the model.

In hindsight, we might have used a readers/writers locking protocol instead of simple locking to synchronize accesses to the DefaultListModel/DefaultTreeModel corresponding to a JListNavigator/JListSortNavigator/JTreeSortNavigator.

Locking the Reduced Model and Other Data Structures The DrJava classes that extend and decorate Swing document classes (AbstractDocument and its descendants) augment the Swing document with extra data structures, most notably the "reduced model" which summarizes key textual features (namely comment and string boundaries and the nesting structure of braces) of the document. These data structures must be protected by essentially the same locking regimen as the embedded Swing document. Whenever these structures are read or modified, the code must first perform a read lock on the associated document (typically this). In addition, the data structure itself must be protected from concurrent access by its own lock because multiple threads can obtain a read lock on the document. The most important auxiliary data structure in our document classes is the reduced model. *When DrJava code accesses the reduced model, it must acquire a read lock or write lock on the associated document before locking the reduced model.* If the associated document is potentially modified by the code for this operation, the lock must be a write lock. Otherwise, it should be a read lock.

All accesses to all DrJava objects accessed by multiple threads must be controlled by a synchronization policy. In most cases, they can be accessed only after a corresponding lock is acquired. In some cases, marking the objects as final or volatile may suffice.

Unsynchronized Read Operations There is an important class of read operations that are exempt from synchronization requirements in Java applications that run on a single processor. If a single read operation is the only observation required to perform an atomic operation on a shared data structure, then it can usually be done without any synchronization. This optimization assumes that the state of this field is always valid, *i.e.*, that it is never given a dummy value that should not be observed while it is being updated. In addition, it assumes that the value of such a field has been fully initialized and that the observing thread can *see* the most recent update of the field that must have happened as constrained by control flow.

A shared variable that is directly assigned to the value of new operation may not be fully initialized because the variable is bound to the location of the allocated object *before* the constructor is invoked for that object. This problem can be avoided by binding the value of the new operation to a temporary variable and assigning the temporary to the shared variable.[2]

The visibility problem is more subtle. The Java memory model allows the Java compiler to cache the values of fields (local variables cannot be shared) in registers (including stack locations) and to re-order the execution of updates to non-volatile fields as long as no dependences are violated (which assures that the semantics of purely sequential code is unaffected). They only have to be written back to the heap (making their updated values visible to other threads) to conform to memory barriers created by synchronization operations and volatile declarations. If accesses from different threads are both protected by a common lock, then the code produced by the compiler must ensure that accesses of these threads are visible to each other. Note that the threads must use the *same* lock. The compiled code does not have to write back the values of fields modified under the protection of a lock when the lock is released. The code can leave the values in registers until an attempt is made to access them under protection of the same lock.

Since locking is an expensive operation, Java provides a lighter weight mechanism called a volatile declaration for ensuring that updates to a field are visible immediately. If a field is declared as volatile, then its value must be immediately written back to the heap every time that it is modified. Hence, all shared fields of Java classes should be declared volatile unless they are already uniformly protected by explicit locking operations. For example, the field _reducedModel in the various DrJava document classes does not need to be volatile because it is always explicitly locked before it is

---

[2] I don't know why this indirect assignment semantics for new operations is not built into Java. Perhaps the designers of Java did not want to foster the development of Java applications that only work correctly when run on a single processor.

accessed. On the other hand, the _current field in JListNavigator must be declared as volatile because accesses to this variable are not systematically protected by a particular lock.. In practice, the failure to declare fields as volatile appears to be masked in most situations in uniprocessor implementations of Java. This is an empirical observation not based on any assurances from the Sun JVM developers. Multiprocessor personal machines will soon be the norm so no application can legitimately be developed assuming that it will be run on a uniprocessor. Moreover, there is nothing in the Java specification that states a uniprocessor implementation has to treat all fields as volatile.

Implications of Multiple Processors Hyperthreaded CPUs that "simultaneously" run two independent threads on the same CPU core are now commonplace in personal computers. Such CPUs can accommodate two different threads in the CPU at once. When one thread stalls because it is waiting for data from main memory or some other reason, the CPU switches instantaneously to running the other thread. We have not observed any of the pathologies associated with multiple processors in hyperthreaded CPUs for reasons that we will discuss later.

Many servers and some high end personal computers have multiple CPU cores. Until recently, all commonly used CPU chips only contained a single core so multicore computers were restricted to architectures accommodating multiple CPU chips. But the most recent wave of higher end CPU chips contain multiple cores, which will soon make multicore computer architectures the norm in personal computers. This change has enormous implications for Java applications including DrJava.

In the context of multicore architectures, unsynchronized read operations are unsafe for two reasons. First, each processor has a separate cache which means that writes performed by one thread may not be visible by another thread in the absence of a protocol that flushes the cache. On a single processor, this flushing takes place on every context switch, so the caching of writes is transparent. On multicore architectures, however, no such serial flushing is possible—making the application program responsible for implementing the necessary cache flushing. As we observed above, the Java Virtual Machine assumes responsibility for writing cached copies of field values to main memory in certain situations involving *explicit* synchronization.[3] In particular, writes and reads that are done under the protection of the *same* lock behave as if they are serially interleaved operations on main memory. In other words, if a write operation happens before a read operation, it will be visible to the read operation. In the absence of synchronization or volatile declarations, writes that happen before reads may not be visible.

Declaring a field as volatile is a lighter weight alternative to explicit synchronization that ensures the atomicity of writes and reads assuming that the field that is no larger than 32 bits (excluding fields of type double and long). The use of volatile declarations only suffices when the desired atomic

---

[3] Local variables in methods are unaffected by this caching process because they are only accessible in the thread executing the method.

operations are reads and writes—properties that we have been taking for granted for *all* fields in the coding of DrJava. We need to make sure that all shared fields are either (i) always protected by synchronization or (ii) declared as volatile. Unfortunately, Java has no mechanism for declaring and enforcing which fields can be shared. It should!

By the way, declaring fields of type long and double is still useful because it ensures that writes are immediately visible to all threads. The only problem is that another thread can see such a field in an inconsistent state when only half of it has been updated! But another thread may be able to deduce that an update of a long or double is complete because a subsequent operation on a volatile variable (that is not long or double) has occurred. The order of accesses to volatile variables within a given thread cannot be changed by the compiler! Moreover, the updates must be made visible to other threads in the order in which they occur.

Besides field caching, there is another potential source of synchronization failure: compiler optimization. The JIT compiler in the JVM is allowed to reorder the execution of read and write operations within a thread as long as the optimization are *transparent to that thread.* Unfortunately, these reorderings may not be transparent to other threads. Explicit synchronization and volatile declarations establish *memory barriers* that prevent the compiler from caching and reordering accesses to fields. For example, the compiler cannot move a read or write operation across a synchronization barrier or reorder read and write operations on volatile variables. In essence, compiler optimizations cannot visibly affect the execution of code involving shared data as long as this code uses either explicit synchronization or final and volatile declarations to protect accesses to shared data.

We have not discussed final fields. Once a constructed object becomes visible to another thread, all of the final and volatile fields in that object must be initialized and visible. Moreover, the final fields hereditarily within such objects are visible. Note that the same claim is not true for fields that are neither final nor volatile. Failing to declare a shared constant field as final is a major coding error because the visibility of its initial value in threads other than the initializing thread can be deferred indefinitely. The JVM can return the value null or some partially initialized object when the field is accessed.

The moral of this story is that in multi-thread programs, the final and volatile modifiers are your friends. The volatile modifier inhibits some optimization but it has a minimal impact on most computations.

In hyperthreaded CPUs, field caching within individual threads has not yet proven to be a problem in DrJava perhaps because the two threads involved share the same cache. We also have not yet seen any code reordering effects but we always execute DrJava on our hyperthreaded machines in "client mode" and suspect that little reordering is done in client mode. We might see different results in "server mode". Some of our dual core machines automatically use "server" mode, which may be one reason why they have produced many more synchronization glitches during testing.

Revising DrJava to Cope with Multiple Processors We clearly need to declare all shared fields that are read or written without synchronization as final or volatile. We also may need to introduce additional explicit synchronization although we have not yet identified any specific examples.

Further Reading on the Java Memory Model and Concurrency We recommend the FAQ web site for Java JSR-133 (the new Java Memory Model) and the sites that it references. The FAQ can be found at: http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html.