Developing with real-time Java, Part 1: Exploit real-time Java's unique features

Take advantage of real-time Java performance in your application

Sean C. Foley

September 01, 2009

Real-time Java™ combines ease of programming in the Java language with the performance required by applications that must conform to real-time constraints. Extensions to the Java language provide features for real-time environments that are lacking in the traditional Java runtime environment. This article, the first in a three-part series, describes some of these features and explains how you can apply them to enable real-time performance in your own applications.

View more content in this series

Real-time Java is a set of enhancements to the Java language that provide applications with a degree of real-time performance that exceeds that of standard Java technology. Real-time performance differs from traditional throughput performance, which is typically a measurement of the total number of instructions, tasks, or work that can be done over a fixed amount of time. Real-time performance focuses on the time an application requires to respond to external stimuli without exceeding given time constraints. In the case of hard real-time systems, such constraints must never be exceeded; soft real-time systems have a higher tolerance for violations. Real-time performance requires that the application itself gain control of the processor so that it can respond to stimuli, and that while responding to the stimuli the application's code is not blocked from execution by competing processes within the virtual machine. Real-time Java delivers responsiveness previously unmet in Java applications.

A real-time JVM can take advantage of real-time operating system (RTOS) services to provide hard real-time capabilities, or it can run on more conventional operating systems for applications with softer real-time constraints. Some of the technologies used in real-time Java come "for free" when you switch to using a real-time JVM. But to exploit certain features of real-time Java, some changes to the application are required. These features are the focus of this article.

Subprocesses that must be constrained

A JVM services a given application by performing work that is only loosely controllable by the application. Several run-time subprocesses are at work within the JVM, including:
- **Garbage collection**: This is the work to reclaim blocks of run-time memory discarded by the application. Garbage collection can delay application execution for a period of time.

- **Class loading**: This process — so called because Java applications are loaded at the granularity of classes — involves loading the application structures, instructions, and other resources from the file system or network. In standard Java the application loads each class when it is first referenced (lazy loading).

- **Just-in-time (JIT) dynamic compilation**: Many virtual machines use dynamic compilation of methods from interpreted Java bytecode into native machine instructions while the application runs. Although this improves performance, the compilation activity itself can cause a temporary delay, blocking application code from running.

- **Scheduling**: In standard Java, a minimal degree of control is permitted to the application to dictate both the scheduling of its own running threads and the scheduling of the application in relation to other applications running on the same operating system.

All of these subprocesses can hamper an application's ability to respond to external stimuli, because they can delay the execution of application code. For instance, a sequence of instructions might be scheduled to execute in response to a signal from a network, a radar system, a keyboard, or any other device. A real-time application has a minimal acceptable period of time in which an unrelated process such as garbage collection is permitted to delay the execution of the responding sequence of instructions.

Real-time Java provides varied technologies designed to minimize interference to the application from these underlying subprocesses. The "for-free" technologies that come when you switch to a real-time JVM include specialized garbage collection that limits the duration and impact of interruptions for collection, specialized class loading that allows for optimized performance at start-time rather than delayed optimization, specialized locking and synchronization, and specialized priority thread scheduling with priority inversion avoidance. However, some modifications to the application are required — particularly to exploit features introduced by the Real-Time Specification for Java (RTSJ).

The RTSJ provides an API enabling numerous real-time features within JVMs. Some of these features are mandatory in an implementation of the specification, others optional. The specification covers the general areas of:

- Real-time scheduling
- Advanced memory management
- High-resolution timers
- Asynchronous event handling
- Asynchronous interruption of threads

**Realtime threads**

The RTSJ defines `javax.realtime.RealtimeThread` — a subclass of the standard `java.lang.Thread` class. On its own, RealtimeThread enables some of the specification's advanced features. For instance, real-time threads are subject to the real-time thread scheduler. The scheduler provides a unique range of scheduling priorities and can implement the first-in, first-out real-time scheduling policy (ensuring that the highest-priority threads run without interruption),
along with *priority inheritance* (an algorithm that prevents lower-priority threads from indefinitely holding a lock required by a higher-priority thread that would otherwise run unimpeded — a situation known as *priority inversion*).

You can explicitly construct instances of `RealtimeThread` in your code. But it's also possible to change your application in a minimal way to enable real-time threading, thereby avoiding significant development effort and the associated costs. Shown here are various examples of ways to enable real-time threading least intrusively and most transparently. (You can [download](https://ibm.com/developerWorks) the source code for all of the article's examples.) These techniques enable an application to exploit real-time threads with minimal effort and allow the application to remain compatible with standard virtual machines.

**Assigning thread type by priority**

Listing 1 shows a block of code that assigns a real-time or regular thread based on a priority value. If it is running on a real-time virtual machine, some threads can be real-time threads.

**Listing 1. Assigning the thread class according to priority**

```java
import javax.realtime.PriorityScheduler;
import javax.realtime.RealtimeThread;
import javax.realtime.Scheduler;

public class ThreadLogic implements Runnable {
    static void startThread(int priority) {
        Thread thread = ThreadAssigner.assignThread(
            priority, new ThreadLogic());
        thread.start();
    }

    public void run() {
        System.out.println("Running " + Thread.currentThread());
    }
}

class ThreadAssigner {
    static Thread assignThread(int priority, Runnable runnable) {
        Thread thread = null;
        if(priority <= Thread.MAX_PRIORITY) {
            thread = new Thread(runnable);
        } else {
            try {
                thread = RTThreadAssigner.assignRTThread(priority, runnable);
            } catch(LinkageError e) {} 
            if(thread == null) {
                priority = Thread.MAX_PRIORITY;
                thread = new Thread(runnable);
            }
        }
        thread.setPriority(priority);
        return thread;
    }
}

class RTThreadAssigner {
    static Thread assignRTThread(int priority, Runnable runnable) {
        Scheduler defScheduler = Scheduler.getDefaultScheduler(); 
        PriorityScheduler scheduler = (PriorityScheduler) defScheduler;
        if(priority >= scheduler.getMinPriority()) {
            return new RealtimeThread(
```
null, null, null, null, null, runnable);
}
    return null;
}

The code in Listing 1 must be compiled with the RTSJ classes. At run time, if the real-time classes are not found, the code catches the `LinkageError` thrown by the virtual machine and instantiates regular Java threads in place of real-time threads. This allows the code to run on any virtual machine, whether real-time or not.

In Listing 1, the method providing the `RealtimeThread` objects is separated into a class of its own. This way, the method is not verified until the class is loaded, which is done when the `assignRTThread` method is first accessed. When the class is loaded, the run-time virtual machine bytecode verifier tries to verify that the `RealtimeThread` class is a subclass of the `Thread` class, which fails with a `NoClassDefFoundError` if the real-time classes are not found.

### Assigning threads using reflection

Listing 2 demonstrates an alternative technique that has the same effect as Listing 1. It starts with a priority value to determine the type of thread desired, instantiating either a real-time or a regular thread based on the class name. The reflective code expects the existence of a constructor in the class that takes an instance of `java.lang.Runnable` as the last argument and passes the null value for all the other arguments.

### Listing 2. Using reflection to assign threads

```java
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class ThreadLogic implements Runnable {
    static void startThread(int priority) {
        Thread thread = ThreadAssigner.assignThread(
            priority, new ThreadLogic());
        thread.start();
    }

    public void run() {
        System.out.println("Running " + Thread.currentThread());
    }
}

class ThreadAssigner {
    static Thread assignThread(int priority, Runnable runnable) {
        Thread thread = null;
        try {
            thread = assignThread(priority <= Thread.MAX_PRIORITY, runnable);
        } catch(InvocationTargetException e) {
        } catch(IllegalArgumentException e) {
        } catch(InstantiationException e) {
        } catch(ClassNotFoundException e) {
        }
        if(thread == null) {
            thread = new Thread(runnable);
            priority = Math.min(priority, Thread.MAX_PRIORITY);
        }
        thread.setPriority(priority);
        return thread;
    }
```
The code in Listing 2 need not be compiled with the real-time classes on the classpath, because
the real-time threads are instantiated using Java reflection.

**Assigning thread type by class inheritance**

The next example illustrates how changing the inheritance of a given class can exploit real-
time threads. You can create two versions of a given thread class, one that is aware of
java.realtime.RealtimeThread and one that is not. Your choice of one or the other might depend
on the underlying JVM. You can enable either one by simply including the corresponding class
file in your distribution. With either choice, the code is relatively simple and avoids any exception
handling, unlike the previous examples. However, when you distribute the application, you must include one of the two class choices, depending on the associated virtual machine that will run the application.

The code in Listing 3 creates regular Java threads in a standard manner:

**Listing 3. Using class inheritance to assign threads**

```java
import javax.realtime.PriorityScheduler;
import javax.realtime.RealtimeThread;
import javax.realtime.Scheduler;

public class ThreadLogic implements Runnable {
    static void startThread(int priority) {
        ThreadContainerBase base = new ThreadContainerBase(priority, new ThreadLogic());
        Thread thread = base.thread;
        thread.start();
    }

    public void run() {
        System.out.println("Running " + Thread.currentThread());
    }
}

class ThreadContainer extends ThreadContainerBase {
    ThreadContainerBase(int priority, Runnable runnable) {
        super(new Thread(runnable));
        if(priority > Thread.MAX_PRIORITY) {
            priority = Thread.MAX_PRIORITY;
        }
        thread.setPriority(priority);
    }
}

class ThreadContainerBase {
    final Thread thread;
    ThreadContainerBase(Thread thread) {
        this.thread = thread;
    }
}
```

To enable real-time threads, you can change the `ThreadContainer` code as shown in Listing 4:

**Listing 4. An alternate thread container class for real-time enablement**

```java
class ThreadContainer extends ThreadContainerBase {
    ThreadContainer(int priority, Runnable runnable) {
        super(assignRTThread(priority, runnable));
        thread.setPriority(priority);
    }

    static Thread assignRTThread(int priority, Runnable runnable) {
        Scheduler defScheduler = Scheduler.getDefaultScheduler();        
        PriorityScheduler scheduler = (PriorityScheduler) defScheduler;
        if(priority >= scheduler.getMinPriority()) {
            return new RealtimeThread(null, null, null, null, null, runnable);
        } else {
            return new Thread(runnable);
        }
    }
}
```
You can include this newly compiled `ThreadContainer` class file in your application instead of the old one when running it with a real-time JVM.

**Segregated memory areas**

Common to all JVMs, including real-time JVMs, is the garbage-collected heap. The JVM reclaims memory from the heap via garbage collection. Real-time JVMs have garbage-collection algorithms specifically designed to avoid or minimize interference with the running application.

The RTSJ introduces the concept of an *allocation context* for each thread, and it introduces additional memory areas. When a memory area serves as the allocation context for a thread, all objects instantiated by the thread are allocated from that area. The RTSJ specifies these additional segregated memory areas:

- The singleton *heap* memory area.
- A singleton *immortal* memory area, from which memory is never reused. The thread initializing a class uses this area as the allocation context when running the static initializer. Although immortal memory requires no attention from the garbage collector, its use is not unlimited, because the memory cannot be reclaimed.
- *Scoped memory areas* (*scopes*). Scopes require no activity from garbage collection, and their memory can be reclaimed — entirely at once — for reuse. The objects allocated in a scope are finalized and cleared, freeing their allocated memory for reuse, when the virtual machine has determined that the scope is no longer the allocation context area for any live thread.
- *Physical* memory areas identified by type or address. You can designate each physical memory area for reuse as a scoped area, or for single use as an immortal area. Such memory areas can provide access to memory with specific characteristics or from specific devices, such as flash memory or shared memory.

Scopes introduce enforced restrictions on object references. When a scoped memory block is released and the objects within are cleared, no object with a reference pointing into the released memory block can exist, which would result in a dangling pointer. This is accomplished in part by the enforcement of assignment rules. The rules dictate that objects allocated from nonscoped memory areas cannot point at scoped objects. This ensures that when the scoped objects are released, objects from other memory areas are not left with references to nonexistent objects.

Figure 1 illustrates these memory areas and assignment rules:
The assignment rules do allow the objects in one scope to point to another. However, this means that there must be an enforced sequence of scope clean-up for each thread, a sequence maintained by a stack within each thread. The stack also includes references to other memory areas that have been entered in addition to scopes. Whenever a memory area becomes the allocation context for a thread, it is placed on top of the thread's scope stack. The assignment rules dictate that objects within scopes higher on the stack can refer to objects within scopes lower on the stack, because the scopes at the top are cleared first. References from lower scopes to higher scopes are forbidden.

The order of scopes on the stack is also coordinated with the order of scopes on the stack of other threads. Once a scope has been placed on the stack of any thread, the scope nearest below it on the stack is considered the parent (or the parent is considered the solitary primordial scope, if no other scope is on the stack). While that scope remains on that stack, it can be placed on the stack of any other thread only if the parent remains consistent, meaning it is the highest scope on the other thread's stack. In other words, a scope in use can have only a single parent. This ensures that when scopes are released, clean-up occurs in a sequential order that's the same regardless of which thread performs the clean-up of each scope, and that the assignment rules maintain consistency across all threads.

**How to exploit segregated memory areas**

You can use a specific memory area by specifying the area as the initial memory area for a thread to run in (when the thread object is constructed), or by explicitly entering the area, providing it with aRunnable object to be executed with the area as the default area.

Special consideration must be taken when you use different memory areas, because they bring with them complications and possible risks. You must choose the size and the number of areas.
If scopes are in use, you must be careful designing the order of the scope stacks of threads, and must remain cognizant of the assignment rules.

**Options for scheduling time-sensitive code**

When you use memory areas other than the heap, you can choose to use `javax.realtime.NoHeapRealtimeThread` (NHRT), a subclass of `javax.realtime.RealtimeThread` that enables threads that are guaranteed to run without interference from the garbage collector. They can run without interference because they are restricted from accessing any object allocated from the heap. Any attempt to violate this access restriction causes a `javax.realtime.MemoryAccessError` to be thrown.

Another scheduling option is the asynchronous event handler, which you can use to schedule code to be executed in response to asynchronous or periodic events. (The events can be periodic if they are initiated by a timer.) This allows you to forgo the need to schedule threads explicitly for such events. Instead, the virtual machine maintains a pool of threads that are shared and dispatched to run the code of asynchronous event handlers whenever events occur. This can simplify real-time applications, freeing you from the management of threads and memory areas.

The class diagram in Figure 2 shows the options available for scheduling code:

**Figure 2. Class diagram illustrating options for scheduling code**

![Class diagram](image)

Figure 3 show how asynchronous event handlers are dispatched:
Figure 3. How asynchronous event handlers are dispatched

Generally, it can be beneficial for portability and modularity to separate the code that responds to the event from the code that enables and dispatches the handler. When the code is encapsulated in an implementation of `java.lang.Runnable`, then a number of options are possible for dispatching that code. You can choose to construct a thread to execute the code, or use asynchronous event handlers that employ pools of threads to execute code on demand, or use combinations of the two.

Table 1 shows a general breakdown of the characteristics of various possible choices:

<table>
<thead>
<tr>
<th>Table 1. Comparison of methods to dispatch code in real-time Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shares threads to execute code</td>
</tr>
<tr>
<td>Regular Thread</td>
</tr>
<tr>
<td>RealtimeThread</td>
</tr>
<tr>
<td>NoHeapRealtimeThread</td>
</tr>
<tr>
<td>AsyncEventHandler</td>
</tr>
<tr>
<td>BoundAsyncEventHandler</td>
</tr>
<tr>
<td>No-heap AsyncEventHandler</td>
</tr>
<tr>
<td>No-heap BoundAsyncEventHandler</td>
</tr>
</tbody>
</table>

Certain design issues unique to real-time Java come into play when you're considering which scheduling options and memory areas to use. Programming for real-time environments in general is a more challenging task than programming straightforward traditional applications, and real-time
Java introduces its own challenges. Table 2 lists some of the complications that can be introduced when additional memory areas, NHRTs, and other real-time features are used:

**Table 2. Some complications and pitfalls of real-time threading and memory areas**

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory allocated to a memory area</td>
<td>Each memory area created by an application is allocated with a requested size. Choosing a size too large is inefficient use of memory, but choosing a size too small can leave the application vulnerable to OutOfMemoryError. During development, even when an application does not change, the underlying libraries can change. This can result in unexpected additional memory usage, causing memory-area limits to be exceeded.</td>
</tr>
<tr>
<td>Timing considerations for shared scopes</td>
<td>A scoped memory area shared by several threads may appear to have sufficient size because it is expected to be cleared when no threads are using it. However, with subtle changes in the timing of threads using the scope, there may never be a time when the scope is unused as the allocation context for any thread. This results in the unexpected possibility that it will never be cleared, causing an OutOfMemoryError. Temporary lock contention between threads can occur when shared scoped areas are entered and cleared.</td>
</tr>
</tbody>
</table>
| The run-time exceptions IllegalAssignmentError, MemoryAccessError, and IllegalThreadStateException | These exceptions can result if insufficient attention is paid to code design. In fact, subtle changes in program behaviour and timing can cause them to appear unexpectedly. Some examples are:  
  • An object from the heap that would normally not be available to an NHRT can become available because of changes in timing and synchronization between threads.  
  • An IllegalAssignmentError can be introduced when it isn't known which memory area an object is allocated from, or where on the scope stack a particular scope is located.  
  • IllegalThreadStateException is thrown when code that enters scoped memory areas is run by regular threads.  
  • Code that makes common use of static fields or other means of caching data is unsafe for scopes because of the assignment rules, which can result in an IllegalAssignmentError. |
| Class initialization                                | Any type of regular or real-time thread can initialize a class, including a NHRT, which can cause unexpected MemoryAccessError. |
| Finalization of objects with the finalize method    | The last thread to exit a scope is used to finalize all objects within:  
  • If finalize methods create threads, scopes may not be cleared as expected.  
  • Finalization can also introduce deadlocks. Prior to finalizing the memory area, the finalizing thread may have acquired locks. Contention for these locks from other threads, and also locks to be acquired during finalization, might occur and can result in deadlock. |
| Unexpected NHRT delays                             | NHRTs, although guaranteed to run without direct interference from garbage collection, can share locks with other types of threads preemtible by garbage collection. If the NHRT is delayed when trying to acquire such a lock while the thread owning the lock is delayed by garbage collection, then the NHRT is indirectly delayed by garbage collection as well. |
A comprehensive example

The next example encompasses some of the real-time features described so far. To start, Listing 5 shows two classes describing a producer of event data and a consumer. Both classes are implementations of Runnable so that they can easily be executed by any given Schedulable object.

Listing 5. Producer and consumer classes for event objects

class Producer implements Runnable {
    volatile int eventIdentifier;
    final Thread listener;

    Producer(Thread listener) {
        this.listener = listener;
    }

    public void run() {
        LinkedList<Integer> events = getEvents();
        synchronized(listener) {
            listener.notify();
            events.add(++eventIdentifier); //autoboxing creates an Integer object here
        }
    }

    static LinkedList<Integer> getEvents() {
        ScopedMemory memoryArea = (ScopedMemory) RealtimeThread.getCurrentMemoryArea();
        LinkedList<Integer> events =
            (LinkedList<Integer>) memoryArea.getPortal();
        if(events == null) {
            synchronized(memoryArea) {
                if(events == null) {
                    events = new LinkedList<Integer>();
                    memoryArea.setPortal(events);
                }
            }
        } return events;
    }
}

class Consumer implements Runnable {
    boolean setConsuming = true;
    volatile boolean isConsuming;

    public void run() {
        Thread currentThread = Thread.currentThread();
        isConsuming = true;
        try {
            LinkedList<Integer> events = Producer.getEvents();
            int lastEventConsumed = 0;
            synchronized(currentThread) {
                while(setConsuming) {
                    while(lastEventConsumed < events.size()) {
                        System.out.print(events.get(lastEventConsumed++) + " ");
                    }
                    currentThread.wait();
                }
            }
        } catch(InterruptedException e) {
        } finally {
            isConsuming = false;
        }
    }
}
In Listing 5, producer and consumer objects access a queue of events that are encoded as a sequence of java.lang.Integer objects. The code expects the current allocation context to be a scoped memory area and expects the queue of events to be stored as the scope's portal object. (The portal is an object allocated from the scope that can be stored in the scoped memory area object itself, a useful convenience because scoped objects cannot be stored in either static fields or in objects allocated from a parent scope.) If the queue is not found, it is created. A couple of volatile fields are used to inform interested threads about the progress of production and consumption of events.

The two classes in Listing 6 show how the code in Listing 5 can be executed:

**Listing 6. Schedulable classes**

class NoHeapHandler extends AsyncEventHandler {
    final MemoryArea sharedArea;
    final Producer producer;

    NoHeapHandler(
        PriorityScheduler scheduler, 
        ScopedMemory sharedArea, 
        Producer producer) {
        super(new PriorityParameters(scheduler.getMaxPriority()), 
              null, null, null, null, true);
        this.sharedArea = sharedArea;
        this.producer = producer;
    }

    public void handleAsyncEvent() {
        sharedArea.enter(producer);
    }
}

class NoHeapThread extends NoHeapRealtimeThread {
    boolean terminate;
    final MemoryArea sharedArea;
    final Consumer consumer;

    NoHeapThread(
        PriorityScheduler scheduler, 
        ScopedMemory sharedArea, 
        Consumer consumer) {
        super(new PriorityParameters(scheduler.getNormPriority()), 
              RealtimeThread.getCurrentMemoryArea());
        this.sharedArea = sharedArea;
        this.consumer = consumer;
    }

    public synchronized void run() {
        try {
            while(true) {
                if(consumer.setConsuming) {
                    sharedArea.enter(consumer);
                } else {
                    synchronized(this) {
                        if(!terminate) {
                            if(!consumer.setConsuming) {
                                wait();
                            }
                        } else {
                            break;
                        }
                    }
                }
            }
        }
    }
}
In Listing 6, the data-producer code is assigned to an asynchronous event handler, to be run at the highest priority available. The handler simply enters a scoped memory area to run the producer code. The same scoped memory area is a parameter to an NHRT class that acts as the consumer of the data. The thread class is also straightforward, allowing synchronized access to `terminate` and `setConsuming` fields to dictate behaviour. When the consumer thread is consuming events, it enters the shared memory area to execute the consumer code, running at a lower priority than the producer. (The consumption behaviour in the example is trivial, simply printing the event identifier to the console.)

Listing 7 shows the code initializing the system and exhibiting the system behaviour:

**Listing 7. System behaviour**

```java
public class EventSystem implements Runnable {
    public static void main(String args[]) throws InterruptedException {
        RealtimeThread systemThread = new RealtimeThread(
            null, null, null, new VTMemory(20000L), null, null) {
            public void run() {
                VTMemory systemArea = new VTMemory(20000L, new EventSystem());
                systemArea.enter();
            }
        }; systemThread.start();

        public void run() {
            try {
                PriorityScheduler scheduler = (PriorityScheduler) Scheduler.getDefaultScheduler();
                VTMemory scopedArea = new VTMemory(20000L);
                Consumer consumer = new Consumer();
                NoHeapThread thread = new NoHeapThread(scheduler, scopedArea, consumer);
                Producer producer = new Producer(thread);
                NoHeapHandler handler = new NoHeapHandler(scheduler, scopedArea, producer);
                AsyncEvent event = new AsyncEvent();
                event.addHandler(handler);

                int handlerPriority = ((PriorityParameters) handler.getSchedulingParameters()).getPriority();
                RealtimeThread.currentRealtimeThread().setPriority(handlerPriority - 1);

                thread.start();
                waitForConsumer(consumer);

                //fire several events while there is a consumer
                event.fire();
                event.fire();
                event.fire();
                waitForEvent(producer, 3);
                setConsuming(thread, false);

                //fire a couple of events while there is no consumer
                event.fire();
                event.fire();

            } catch(InterruptedException e) {} 
        }
    }
}
```

Developing with real-time Java, Part 1: Exploit real-time Java's unique features
In Listing 7, a pair of scopes is used as the base for the scope stack for the no-heap thread and handler, a requirement because these **Schedulables** cannot access any object that is heap-allocated. An asynchronous event object represents the event, with the attached handler to be dispatched when the event is fired. Once the system is initialized, the code starts the consumer thread and fires the event several times, running at a priority just below the priority of the event handler. The code also switches off and on the consumer thread while additional events are fired.

Listing 8 shows the output when **EventSystem** runs in a real-time JVM:

**Listing 8. Console output**

1 2 3 6

An interesting aspect of this example is the reason why events 4 and 5 are not reported. Each time the listening thread reports the events in the queue, it starts from the front of the queue and goes to the end, suggesting that all six events would be reported at least once.

However, the design ensures that the memory used to store the events is automatically discarded when no thread is consuming them. When a consumer thread stops reading from the queue, it
exits the scoped memory area, at which time no `Schedulable` objects are using the area as the allocation context.

The absence of `Schedulable` objects using the area means the scoped area is cleared of objects and reset. This includes the portal object, so the queue and all events within it are discarded when the thread stops listening. Each time a subsequent event is fired, the queue is recreated and repopulated, but with no listening thread, the memory is discarded immediately afterwards.

The memory management is automatic and runs without interference from the garbage collector, if the collector is active (because the handler and thread are both no-heap). The events are stored as a queue of objects in memory, continuing to grow if a listening thread is available to consume them. If not, the queue and the associated events are automatically discarded.

**A general usage scenario**

With the scheduling and memory-management framework, you can design an application with various priority levels for threads to perform optimally in a real-time virtual machine (and potentially adequately in other virtual machines). The application might include event-handling threads of high-priority levels, collecting data from external inputs and storing the data for processing. Because of their transient and asynchronous nature, these event-handling threads might be suited for alternative memory management, and they may be most critically subject to real-time constraints. At an intermediate priority level there might be processing threads that consume the data and make calculations, or distribute the data. The intermediate threads might require adequately allocated CPU utilization to manage their workloads. At the lowest priority levels, there might be maintenance and logging threads. Using a real-time virtual machine to manage the scheduling and memory usage of these various tasks in the application can allow it to run most efficiently.

The RTSJ’s intent is to enable developers to write applications that run within required real-time constraints. Simply using the real-time scheduler and threads can be enough to realize that goal. If not, more-advanced development may be necessary to take advantage of one or more of the more-advanced features implemented by the virtual machine.

**Conclusion to Part 1**

This article has outlined some tips to get you started integrating elements of real-time Java into your Java application. It covers some of the scheduling and memory-management features that you might wish to use to realize real-time performance. This is a starting point for you to leverage the traditional benefits of the Java language, such as interoperability and safety, and combine them with new features allowing you to meet the real-time constraints your application requires.

In the next installment in this series, you’ll learn techniques for porting an existing application to real-time Java. The final article will build on the first two parts and take you through designing, validating, and debugging a real-time system that incorporates real-time Java.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code for the article examples</td>
<td>j-devrtj1.zip</td>
<td>5KB</td>
</tr>
</tbody>
</table>
Related topics

- **Real-time Java**: Read this six-part developerWorks series on real-time Java.
- **Real-Time Specification for Java**: You'll find the RTSJ at the Java Community Process site.
- **IBM® WebSphere® Real Time**: Download an evaluation version.
- **IBM Real Time Application Execution Optimizer for Java**: Optimize and verify your real-time Java application for specialized environments with this tool.
- **Gosling: Realtime Java** (Builder.com): In this video, James Gosling dismisses the notion that real-time means real fast.

© Copyright IBM Corporation 2009
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)