



Creating Real-time Applications in the Java Programming Language

by Chris McKillop
Software Engineer
QNX Software Systems Ltd.
cdm@qnx.com

Introduction

Traditionally, developers have used Java to create web-based applications and enterprise-level systems. Yet, many of Java's strengths can also be applied to real-time and embedded applications. Take portability, for instance. By abstracting both the underlying CPU and the underlying OS, Java makes it easier to port application code across product lines, allowing designers to choose the best hardware and system software for any given project. Unfortunately, this same abstraction can also make Java difficult to use in some embedded environments.

To address such problems and enhance Java's suitability for embedded real-time systems, a committee of experts from Sun, QNX, Nortel, and other organizations developed the Real-time Specification for Java, or RTSJ. The committee, called the Real-Time for Java Expert Group, submitted their Java Specification Request, or JSR, as part of the Java Community Process. The JSR was ratified and subsequently implemented in two commercial Java virtual machines: IBM's J9 and TimeSys's jTime. In fact, jTime serves as the reference implementation of the standard and can be downloaded from the TimeSys website.

The RTSJ had a primary goal: satisfy the needs of real-time systems without breaking any existing Java code. To achieve this goal, it addressed several areas, including:

- memory management
- threads and scheduling
- signals and timers

In some cases, the RTSJ extended the existing API; in other cases, it had to implement a new API. This paper covers each of the above areas, explaining the issues involved, describing how RTSJ solved them, and providing working code examples.

Memory Management

Of all the Java enhancements made by the RTSJ, the new classes for managing memory are probably the most radical. Since other parts of the standard rely on these classes, it is best to review them first.

With a standard Java virtual machine (JVM), the programmer doesn't even have to think about memory — which explains, in large part, why Java is so popular. Since the existing memory management scheme couldn't simply be removed (doing so would break existing code), the RTSJ introduced a new group of classes: Memory Areas.

A standard Java environment uses a single memory area that is “hidden” from the programmer and managed by the garbage collector. Whenever the programmer uses **new** to create an object, memory is taken from the hidden memory area. When that memory is no longer referenced by anything in the system, it is freed by the garbage collector. In a system using RTSJ extensions, object creation is performed in the exact same manner, but now the programmer can control which memory area the **new** operator uses to construct objects, and in turn, how that memory is reclaimed.

To implement the new memory areas, the RTSJ provides a series of abstract base classes. The top-level class, `MemoryArea`, describes the basic interfaces that all memory areas will provide. `ScopedMemory` then extends this new class to provide a new base class for memory areas that reclaim memory when the current scope is exited (similar to stack memory in the C language). All of the memory areas have rules dealing with what other types of memory they can hold references to.

Extending the `MemoryArea` class

The RTSJ provides two classes, `HeapMemory` and `ImmortalMemory`, to extend `MemoryArea` directly. Both classes are singletons (only a single object of the given type can be created in the system), and the single instance can be obtained by invoking the associated `instance()` static function.

- **HeapMemory** — Provides an interface to the standard, garbage collector-managed, Java heap. `HeapMemory` allows code running in another memory context to be allocated objects in the standard Java heap; it also allows generic code to use any memory area given to that code.
- **ImmortalMemory** — Provides a new type of memory area in which allocations exist longer than the application itself; the allocations are reclaimed only when the program completes. Even if no references to the allocated object exist, the object itself will continue to exist. The garbage collector can scan Immortal memory for references, but isn't allowed to change it. Objects stored in the Immortal area may only reference objects allocated from `HeapMemory` or `ImmortalMemory` areas.

Controlling memory areas

The RTSJ also provides two classes, `VTMemory` and `LTMemory`, to extend the `ScopedMemory` base class. The VT and LT designations, which stand for variable time and linear time, indicate how the memory area will behave during allocations:

- **VTMemory** — Makes no assurances as to how long the memory allocation will take, or about the contiguousness of the memory returned between invocations of **new**.

- **LTMemory** — Ensures that the allocation time will be linear and that the initial memory size, specified when the memory area is constructed, will be contiguous. As a result, LT memory areas must preallocate the minimum size specified when constructed, whereas VT memory is free to allocate as needed.

Aside from these runtime differences, both types of memory are used in the same way and both keep an internal reference count for each entry into their area. When the count reaches zero, all allocated objects are freed.

Real-time Threads

To facilitate real-time behavior, the RTSJ introduces two new thread types: `RealtimeThread` and `NoHeapRealtimeThread`. Both types allow programs to create threads that run at a priority higher than the garbage collector. Nonetheless, the types differ when it comes to preempting the garbage collector:

- **RealtimeThreads** — Can prevent the garbage collector from running, but won't preempt the collector if it is already running.
- **NoHeapRealtimeThreads** — Will preempt a running garbage collector in all cases.

Both threads extend the standard Java `Thread` class, allowing the existing API to work as expected. Unlike standard threads, RTSJ threads take a variety of new parameters, allowing the thread creator to control both the scheduling and the memory areas of the thread.

Realtime thread scheduling

Standard Java Threads are scheduled in a fixed priority scheme, with the definition of scheduler behavior being loosely defined. The RTSJ provides a more formal definition of scheduler behavior, of the objects that can be scheduled, and of how they are controlled. According to the specification, a compliant system must have a base scheduler (`PriorityScheduler`) that provides least 28 “real-time” priority levels in addition to the 10 standard Java priority levels. If multiple threads become ready to run, the base scheduler will schedule the highest-priority thread for execution — much like the schedulers found in commercial RTOSs such as the QNX[®] Neutrino[®] OS or VxWorks.

Developers can create instances of `RealTimeThreads` just as they would standard Java threads, and the system will provide reasonable defaults for all of the extended parameters. The same isn't true of `NoHeapRealtimeThreads`, however. Since these threads can't access the Java heap: a) they must be created with either an `ImmortalMemory` or `ScopedMemory` area to use for memory allocations; and b) the garbage collector never has to worry about the memory they reference. This explains why the `NoHeapRealtimeThread` can unconditionally preempt the garbage collector.

Code example

The following code example shows how the new thread types and new memory areas work together. In this example, the same test code executes twice — the first time using a `ScopedMemory` area and then again using `HeapMemory`. When run with the `ScopedMemory`, the pre-, inner- and post-loop consumed values will always be 0, since the execution scope for the memory area is fully contained within the `run()` method of the `memoryLoop` object. When run with `HeapMemory`, the consumed memory value will grow linearly with the test iterations until the garbage collector runs and cleans up the heap.

```

import javax.realtime.*;

public class ScopedMemoryExample {
    //
    // Chew up some memory when run. In a normal Java environment this function
    // would use ~4K per loop and it would be up to the garbage collector to
    // clean up the unreferenced memory. When run inside of a ScopedMemory
    // area, the memory will be cleaned up when the run() is complete.
    //
    public static Runnable memoryLoop = new Runnable() {
        public void run() {
            for (int i = 0; i < 5; i++) {
                int foo[] = new int[1024];
                MemoryArea memory = MemoryArea.getMemoryArea(foo);
                System.err.println(" MemoryLoop: consumed: " + memory.memoryConsumed());
            }
        }
    };

    //
    // Use this Runnable as the logic for a RealtimeThread to run our tests.
    //
    public static Runnable threadEntry = new Runnable() {
        private void loop(MemoryArea memory) {
            System.err.println("pre-loop consumed: " + memory.memoryConsumed());
            for (int i = 0; i < 2; i++) {
                memory.enter(memoryLoop);
                System.err.println("inner-loop consumed: " + memory.memoryConsumed());
            }
            System.err.println("post-loop consumed: " + memory.memoryConsumed());
        }
        public void run() {
            System.err.println("Test1: Using a ScopedMemory Area.");
            loop(new VMemory(0, 1024 * 1024));
            System.err.println();
            System.err.println("Test2: Using a HeapMemory Area.");
            loop(HeapMemory.instance());
        }
    };

    //
    // The thread that calls main() will always be a standard Java Thread,
    // so a new RealtimeThread needs to be started to perform our tests.
    //
    public static void main(String[] args) {
        RealtimeThread thread = new RealtimeThread(null, null, null, null, null, threadEntry);
        thread.start();
        try {
            thread.join();
        } catch (InterruptedException e) {
        }
    }
}

```

Figure 1 — ScopedMemory and RealtimeThread example.

Timers and Asynchronous Events

Many real-time systems use a periodic control or processing loop, in which periodic events are managed through asynchronous events or callbacks. For instance, in POSIX systems, developers often use signals to interrupt control flow so as to perform time-critical operations. The RTSJ has several new interfaces and classes that add similar functionality to Java. The generic `AsyncEvent` and `AsyncEventHandler` provide the

base classes on which the asynchronous system is built. In a nutshell, `AsyncEvent` allows one or more `AsyncEventHandlers` to be registered so that their associated `handleAsyncEvent()` methods are invoked when the event is fired.

Timer Classes

The RTSJ's timer classes represent the most basic use of this asynchronous system. Using an instance of these classes, a programmer can set up a callback to occur based on an initial starting time, either absolute or relative, and, depending on the type of timer, a timer period once the timer starts.

The RTSJ defines two timers:

- **OneShotTimer** — Triggers an event once.
- **PeriodicTimer** — Continues to fire events, once started.

See Figure 2 for an example of how to use these timers.

```
import javax.realtime.*;

public class PeriodicTimerExample {
    public static final int TIMEOUT_MS = 100;
    public static AsyncEventHandler handler = new AsyncEventHandler() {
        public void handleAsyncEvent() {
            System.err.println("Timer Fired!");
        }
    };
    public static void main(String[] args) {
        PeriodicTimer timer = new PeriodicTimer(new RelativeTime(TIMEOUT_MS,0),
                                                new RelativeTime(TIMEOUT_MS,0), handler);

        timer.start();
        try {
            //
            // Allow 10 fires before shutting the timer down.
            //
            Thread.sleep(TIMEOUT_MS*10);
        } catch( InterruptedException e ) {
        }
        timer.destroy();
    }
}
```

Figure 2 — PeriodicTimer Example

Periodic Thread Pattern

In addition to a purely asynchronous callback-style timer, the RTSJ provides for a periodic thread pattern in which a real-time thread blocks, waiting for the timeout to occur. The blocking function also informs the calling thread if it has missed any deadlines — that way, the thread can possibly adjust operational values (timer period, iterations per timer, etc.) to make all future deadlines. The following code example shows how to achieve this timer pattern.

```

import javax.realtime.*;

public class PeriodicTimerExample {
    public static final int TIMEOUT_MS = 100;
    public static AsyncEventHandler handler = new AsyncEventHandler() {
        public void handleAsyncEvent() {
            System.err.println("Timer Fired!");
        }
    };
    public static void main(String[] args) {
        PeriodicTimer timer = new PeriodicTimer(new RelativeTime(TIMEOUT_MS,0),
                                                new RelativeTime(TIMEOUT_MS,0), handler);

        timer.start();
        try {
            //
            // Allow 10 fires before shutting the timer down.
            //
            Thread.sleep(TIMEOUT_MS*10);
        } catch( InterruptedException e ) {
        }
        timer.destroy();
    }
}

```

Figure 3 — PeriodicThread Example

Exploring the Possibilities

This paper has only scratched the surface of what the RTSJ standard provides. For instance, the RTSJ also offers:

- Several additional areas that enable direct access to physical memory — useful for talking to devices and for interacting with memory shared among other processes in the system.
- An asynchronous event model that provides a clean *interrupt()* system for threads so that status polling can be avoided.
- A class that lets you use AsyncEvent handlers for POSIX/UNIX signals.
- A series of inner-thread communication data structures that allow real-time and standard threads to communicate with one another.
- A complete system for resource allocations that enables feasibility analysis for more complex thread schedulers.

I invite anyone whose interest has been pricked to visit the RTSJ homepage (<http://www.rtsj.org>), read the standard, obtain a JVM that implements it, and explore the possibilities that a real-time system built with Java can provide.