



The Real-Time Specification for Java™

Greg Bollella, Ph.D.
Senior Staff Engineer
Sun Microsystems Laboratories
greg.bollella@east.sun.com

3/26/01

1

Outline

- ★ The Real-Time Specification for Java
 - ★ The Java Community Process
 - ★ JSR-1 expert group
 - ★ Guiding Principles
 - ★ Scheduling
 - ★ Memory Management
 - ★ Synchronization
 - ★ Asynchronous Event Handling
 - ★ Asynchronous Transfer of Control
 - ★ Asynchronous Thread Termination
 - ★ Physical Memory Access

The Java Community Process

- ✦ A community-wide, software development method for extending Java APIs
- ✦ Java specification request
- ✦ Engineers nominated by participating organizations
- ✦ Lead participant named
- ✦ Specification lead forms expert group
- ✦ Specification developed
- ✦ Participant review
- ✦ Public review
- ✦ Reference implementation and test suites

JSR-1 EG Primary Team

- ✦ Greg Bollella, Sun Labs (formerly IBM)
- ✦ Ben Brosgol, Aonix
- ✦ Peter Dibble, Microware
- ✦ Steve Furr, QSSL
- ✦ James Gosling, Sun Microsystems
- ✦ David Hardin, aJile Systems
- ✦ Mark Turnbull, Nortel Networks

JSR-1 EG Consultants

- ★ Apogee, Wolfgang Pieb
- ★ Carnegie Mellon, Raj Rajkumar
- ★ Lockheed Martin, Doug Locke
- ★ Lucent, Larry Rau
- ★ MITRE, E. Douglas Jensen
- ★ Mitsubishi Electric, Masahiro Kuroda
- ★ Motorola, Edward Wentworth
- ★ NSICom, Alexander Katz
- ★ NIST, Alden Dima
- ★ Rockwell- Collins, Ray Kaman
- ★ Schneider, Rudy Belliardi
- ★ Thomson- CSF, Jean- Michel Meignien
- ★ Wind River, Currently Unassigned
- ★ Honorary, Russ Richards
- ★ Emeritus, George Malek
- ★ Emeritus, Chris Yurkowski
- ★ Emeritus, Mike Schuette
- ★ Emeritus, Simon Waddington

Guiding Principles

- ✦ Temporally Predictable Execution
- ✦ Support Current Real-Time Application Development Practice
- ✦ Backward Compatibility
 - The RTSJ maps the Java Language Semantics (JLS) semantics to appropriate, required instances
- ✦ RTSJ Appropriate for any Java™ Platform
- ✦ WOCRAC (like WORA but different)
 - Write Once Carefully Run Anywhere Conditionally
- ✦ Support Leading Edge RT Application Development and the Real-Time Scheduling Academic Community
- ✦ No Syntactic Extension
- ✦ Allow for Implementation Trade-offs
 - Toys to Cruise Missiles
 - Incentive for RTOS vendors

Scheduling

☀ Scheduler

- Abstract base class
- Contains methods for feasibility analysis, admission control, dispatching, and asynchronous event handling mechanism
- Can be considered distinct from the dispatcher

☀ *Schedulable*

- An Interface
- Any object implementing *Schedulable* is scheduled by a Scheduler
- In the RTSJ `RealtimeThreads` and `AsyncEventHandlers` implement *Schedulable*
- The RTSJ encourages implementations to extend the notion of a schedulable object
- Each *Schedulable* object has a reference to a Scheduler

Scheduling

- ✦ PriorityScheduler extends Scheduler
 - ✦ Required scheduler, i.e., this function will be available on all implementations of the RTSJ
 - ✦ Actually more like a dispatcher
 - ✦ Fixed-priority, preemptive
 - ✦ Priority assignment by application logic
 - ✦ At least 28 *unique* priority levels for RealtimeThreads
- ✦ E.g., RMAScheduler extends Scheduler
- ✦ E.g., EDFScheduler extends Scheduler

Scheduling

- ☀ `RealtimeThread` extends `Thread`

- Managed by a scheduler
- May use memory other than the heap
- Participate in asynchronous transfer of control and thread termination
- May access physical memory

- ☀ `NoHeapRealtimeThread` extends `RealtimeThread`

- Not allowed to read or write to objects on the heap
- Not allowed to manipulate references to objects on the heap
- Must be created with a scoped memory area
- May immediately preempt the garbage collector

Scheduling

☀ SchedulingParameters

- Abstract base class for eligibility metric
- PriorityParameters
 - Traditional priority
 - ImportanceParameters
 - Importance field for overload situations

☀ ReleaseParameters

- Abstract base class for release characteristics
- PeriodicParameters
- AperiodicParameters
- SporadicParameters

☀ MemoryParameters

- Defines a schedulable object's memory demands

☀ ProcessingGroupParameters

- Used to manage many aperiodic or sporadic threads as a meta-level periodic thread

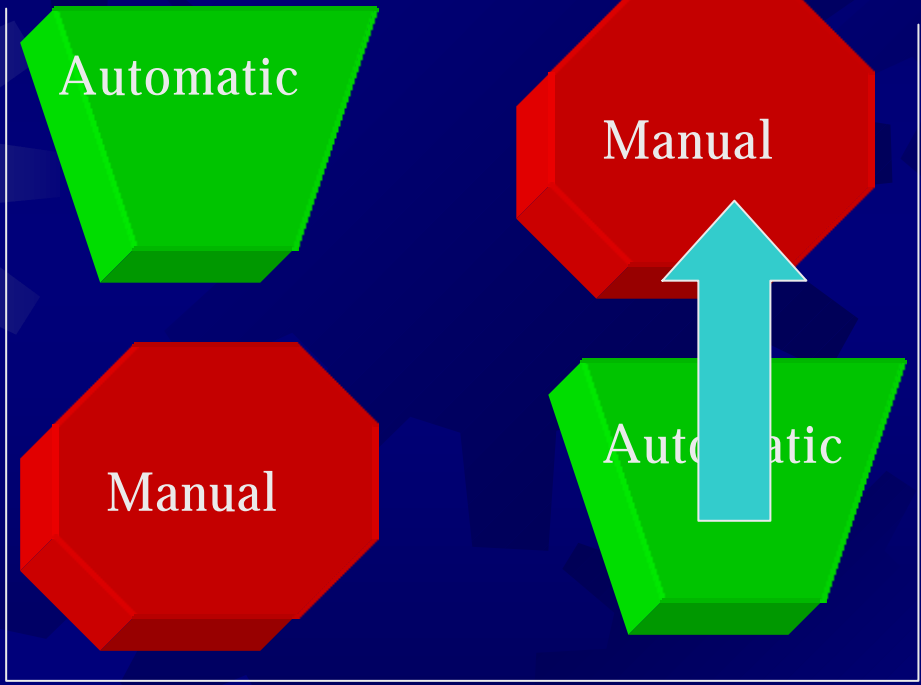
Memory Management

- ★ We note that the JLS is curiously silent on the subject of automatic memory reclamation (aka garbage collection)
- ★ Saying anything about gc seemed to require saying more than the Java™ Programming Language inventors wanted to say
- ★ The JLS allows programmatic allocation of memory (new) but has no programmatic way to deallocate memory
- ★ The RTSJ is also mostly silent on the matter of garbage collection

Memory Management

Increasing

Safety

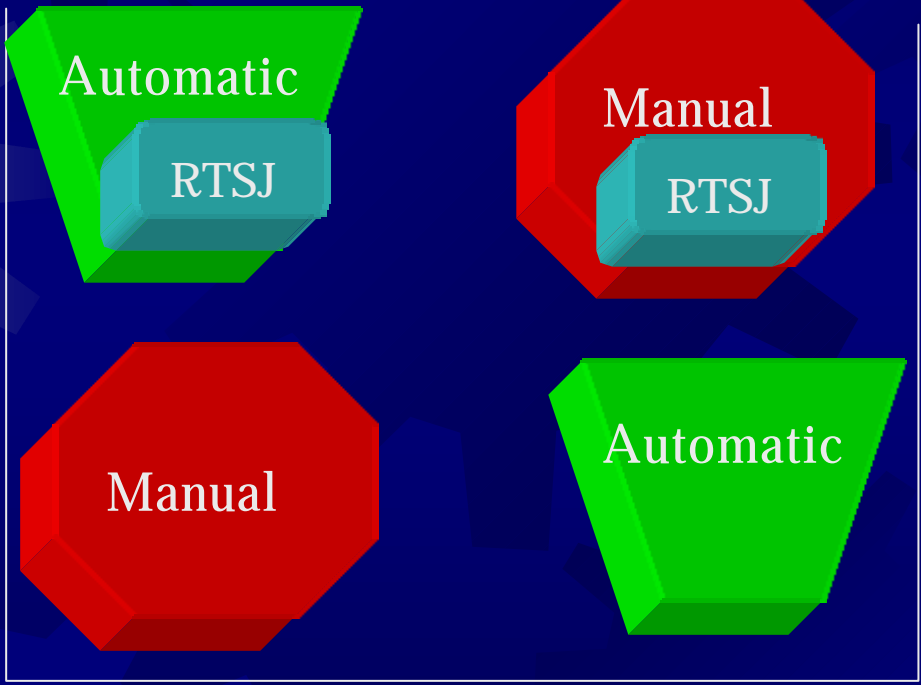


Predictability

Memory Management

Periautomatic

Increasing Safety



Predictability

Increasing

Memory Management

- ★ RTSJ changes the notion of object lifetime (i.e., when an object is a candidate for collection)
- ★ Manual: Lifetime controlled by program logic
- ★ Automatic: Lifetime controlled by visibility
- ★ RTSJ Memory Types: Lifetime controlled by syntactic scope
 - ★ Objects live until control flows out of scope
 - ★ When control leaves scope finalizers execute and complete before the memory area is accessed

Memory Management

☀ Memory Areas

- ☀ Objects not managed by collector

☀ Immortal Memory Area

- ☀ One ImmortalMemory object per JVM™
- ☀ Pre-allocated at JVM start
- ☀ Effective scope is larger than the program, i.e., no control in the program can ever leave the scope of the immortal memory area
- ☀ Used for sharing between real-time threads and sharing between real-time and non-real-time threads

Memory Management

☀ Scoped Memory Areas

- Associated with one or more scopes (closure or thread)
- Scopes may have more than one associated memory area with one primary (where objects are created by default)
- LTMemory - execution time of new is linear in object size
- VTMemory - execution time of new is variable

Memory Management

- ★ Assignment Rules, based on object lifetimes

- Heap ↔ Heap

- Heap ↔ Immortal

- Collector can traverse immortal area and be safely preempted thus we can allow object in the immortal memory area to hold references to objects in the heap

- Immortal ↔ Immortal

- Scoped → Immortal

- Scoped → Scoped (in outer or same scope)

- ★ Partial static analysis for assignment safety is possible (classfiles so marked)

- ★ Runtime checks necessary for unanalyzed or unanalyzable code

Synchronization

- ✦ Priority Inversion Control
- ✦ Default behavior of synchronized must be that of the priority inheritance algorithm
- ✦ Other priority inversion avoidance algorithms can be set for either all or particular monitors
- ✦ Synchronized problematic between regular Java threads and real-time threads
- ✦ NoHeapRealtimeThreads have implicit execution priority higher than the collector
 - ✦ Correct implementation of any priority inversion avoidance algorithm is impossible if execution priority of NHRT is honored
 - ✦ The RTSJ provides three Wait-free Queue classes

Wait-free Queues

- ✦ Unidirectional data flow and non-blocking read/write methods
- ✦ The `write()` method of the `WaitFreeWriteQueue` is the 'real-time' end
- ✦ Wait-free write queue
 - ✦ Number of entries fixed at creation time
 - ✦ Internal objects are allocated from appropriate memory area
 - ✦ Real-time writer does not block on queue-full or queue-empty conditions (instead: application logic determines action (toss, overwrite, etc.))

Asynchronous Event Handling

- ★ Real-time and Embedded Systems are typically tightly coupled to the REAL-WORLD
- ★ Events in the real-world are asynchronous to program execution
- ★ Asynchronous events may also arise internally within the JVM™ (i.e., programmatically)
- ★ The RTJS provides a mechanism to bind a schedulable object to the occurrence of an event
- ★ When the event occurs the object's run state changes to ready-to-run and is scheduled wrt its parameter objects
- ★ Mechanism designed for tens of thousands of events and handlers, i.e., very lightweight

Asynchronous Event Handling

- ★ AsyncEvent
- ★ AsyncEventHandler implements Schedulable
- ★ An instance of AsyncEvent represents something that can happen
- ★ An instance of AsyncEventHandler has a method (`handleAsyncEvent()`) which contains the logic that should execute when the event occurs
- ★ Handlers are bound to events by
 - `AsyncEvent.addHandler(AsyncEventHandler a);`

Asynchronous Event Handling

- ★ An instance of `AsyncEvent` may be bound to an external event using
 - `AsyncEvent.bindTo(String s)`
- ★ There are two ways `AsyncEvents` occur
 - The method `AsyncEvent.fire()` is invoked or
 - an external event occurs
- ★ The execution of handlers is required to be semantically equivalent, wrt scheduling, to instances of `RealtimeThread`

Asynchronous Transfer of Control

- ✦ The Real-Time for Java™ Consultants requested the RTSJ include a concept for allowing the asynchronous transfer of the flow of execution to some predetermined, syntactically defined point in the program
- ✦ The ATC mechanism is similar to exception handling in the JLS (Java exceptions are synchronous).
- ✦ The prime directive for ATC (from ourselves) is:
 - ✦ *Code written without a priori knowledge of possible interruption must not be interrupted*
 - ✦ How does the RTSJ accomplish the prime directive?

Asynchronous Transfer of Control

- ★ `AsynchronouslyInterruptedException`
 - ✱ Only the code within a method with AIE in its throws clause is interruptible
 - ✱ `Timed`
- ★ *Interruptible* (an interface)
 - ✱ Classes which implement can be given to `Timed` constructor

Asynchronous Transfer of Control

- ★ How does logic asynchronously transfer control?
 - `javax.RealtimeThread.interrupt()` has additional semantics
 - when `t.interrupt()` is executed an AIE is thrown at thread `t` and then if:
 - control is in any method with AIE in its throws clause then control will transfer to the calling method with an AIE
 - control is in any method without AIE in its throws clause or in any synchronized block/method the method or block will complete normally and the AIE is set to pending

Interruptible I/O Methods

- ★ The consultants required that the RTSJ should allow a mechanism which would preclude indefinitely blocked I/O calls.
- ★ Methods in `java.io.*` now throw `IOException`, *however, it's typically not implemented.*
- ★ Two cases:
 1. The device (and thus its stream) is no longer needed (or the device no longer exists).
 2. Timed, non-blocking I/O calls (when the device and its associated streams remain viable).

Interruptible I/O Methods

- ★ Case 1: Device no longer needed or gone.
 - Semantics of `stream.close()` and the I/O methods are required to be modified.
 - Blocked I/O calls are *required* to throw appropriate instances of `IOException` when `stream.close()` is called on the stream on which they are blocked.



Interruptible I/O Methods

- ★ Case 2: Timed, non-blocking I/O calls for devices and their streams which remain viable.
 - ★ Programming pattern
 - ★ A simple non-timed, non-blocking I/O call can be easily built from two AsyncEvents and their handlers.

Non-Blocking I/O

```
handleAsyncEvent () {  
    // handler for ae1  
    c = stream.read();  
    // handle IOException  
    // put c somewhere  
    ae2.fire();  
}
```

```
nonblockingRead () {  
    //setup, etc.  
    ae1.fire();  
}
```

```
handleAsyncEvent () {  
    // handler for ae2  
    // get c  
    // do something with c  
}
```

Asynchronous Thread Termination

- ✦ To asynchronously terminate a thread is a requirement from the consultants
- ✦ Arbitrary thread termination is as unsafe as is arbitrary asynchronous transfer of control thus the same prime directive applies
- ✦ ATT typically implies that logic can cause a thread to terminate when some external happening occurs
- ✦ The RTSJ allows ATT by use of the asynchronous event handling and asynchronous transfer of control mechanisms

Physical Memory Access

- ✦ Requirement by consultants and industry input
- ✦ Generalized abstraction of such access is beyond the scope of the charter of the Real-Time for Java Expert Group (RTJEG) (actually, we thought that we did not really know enough about all of the various memory types to create a useful abstraction)
- ✦ The RTJEG chose to specify a low-level mechanism useful for building higher-level abstractions

Physical Memory Access

- ☀ MemoryArea
 - ☀ ImmortalMemory
 - ☀ ImmortalPhysicalMemory
 - ☀ ScopedMemory
 - LMemory
 - ScopedPhysicalMemory
 - VMemory
 - ☀ PhysicalMemoryFactory
 - ☀ RawMemoryAccess
 - ☀ RawMemoryFloatAccess

Physical Memory Access

- ✦ Two styles of access
- ✦ Ability to set and get bytes of physical memory
 - ✦ Useful for device control
 - ✦ `RawMemoryAccess`
 - ✦ `RawMemoryFloatAccess`
- ✦ Ability to allocate objects in physical memory
 - ✦ Programmer managed object cache
 - ✦ `ImmortalPhysicalMemory`
 - ✦ `ScopedPhysicalMemory`
- ✦ Programmers use the physical memory factory to create instances of the three classes

Summary

- ★ The RTSJ addresses seven areas: Scheduling, Memory Management, Synchronization, Asynchronous (Event Handling, Transfer of Control, Thread Termination), and Physical Memory Access
- ★ Current version always available at www.rti.org
- ★ Comments to: comments@rti.org
- ★ “The Real-Time Specification for Java”, Addison-Wesley, June 2000
- ★ Reference implementation target mid-2001

Code Examples

- ★ `RealtimeThread`
- ★ `PeriodicThread`
- ★ `Scheduler`
- ★ `ScopedMemory`
- ★ `AsyncEvent`
- ★ `Timer`
- ★ `AsynchronouslyInterruptedException`

RealtimeThread

```
public class ReceiveThread extends RealtimeThread {
    public void run() {
        /* logic for receive thread */
    }
    public void example() {
        RealtimeThread rt = new ReceiveThread();
        if (!rt.getScheduler().isFeasible())
            throw new Exception("Whatever...");
        rt.start();
    }
}
```

PeriodicThread

```
public class PeriodicThread extends RealtimeThread {  
    PeriodicThread(MyPeriodicParameters pp,  
                  MemoryParameters mp, Runnable r) {  
        super(pp.sp, pp, mp, null, null, r);  
    }  
}
```

Periodic Thread

```
public MyPeriodicParameters(RelativeTime period,  
                             RelativeTime cost) {  
    super(null, /* no start time */  
          period,  
          cost,  
          null, /* deadline == period */  
          null, /* no overrun handler */  
          null); /* no miss handler */  
    sp = new PriorityParameters(determinePriority());  
}}
```

PeriodicThread

```
RealtimeThread rt = new PeriodicThread(  
    new MyPeriodicParameters(new RelativeTime(50, 0),  
                             determineCost()),  
    new Runnable() {  
        public void run() {  
            RealtimeThread t;  
            try {  
                t =  
                    (RealtimeThread)Thread.currentThread();  
                do {  
                    /* thread logic. */  
                } while (t.waitForNextPeriod());  
            } catch (ClassCastException e) {}  
        }  
    });
```

Finding a New Scheduler

```
public class SchedulerExample {
public static Scheduler findScheduler(String policy) {
    String className = System.getProperty(
        "javax.realtime.scheduler." + policy);
    Class clazz;
    try {
        if (className != null
            && (clazz = Class.forName(className)) != null){
            return (Scheduler)clazz.getMethod(
                "instance",null).invoke(null,null);
        }
    } catch (/* lots of exceptions */) {
        return null;
    }
}
```

Finding a New Scheduler

```
Scheduler scheduler = findScheduler("EDF");
if (scheduler != null) {
    RealtimeThread t1 = new RealtimeThread(null,
        new PeriodicParameters(
            null, new RelativeTime(100, 0),
            new RelativeTime(5, 0),
            new RelativeTime(50, 0), null,
            null),
        null, null, null, null) {
        public void run() {
            /* thread processing */
        }
    };
    t1.setScheduler(scheduler);
    t1.start();
}
```

ScopedMemory

```
final ScopedMemory scope = new LTMemory(1024, 16 * 1024);
scope.enter(new Runnable() {
    public void run() {
        /* Do some time-critical operations */
        try {
            /* To allocate from the heap */
            HeapMemory.instance()
                .newInstance(Class.forName("Foo"));
            /* Allocate from the previous scope*/
            scope.getOuterScope()
                .newInstance(Class.forName("Foo"));
        } catch (ClassNotFoundException e) {
        } catch (IllegalAccessException ia) {
        } catch (InstantiationException ie) {
        }
    }
});
```

ScopedMemory

```
final ScopedMemory scope =  
    new LTMemory(0, 16 * 1024);  
RealtimeThread t1 =  
    new RealtimeThread(null, null,  
        new MemoryParameters(100000, 0), scope, null,  
        new Runnable() {  
            public void run() {  
                /* do some stuff */  
            }  
        }  
    );
```

AsyncEvent

```
try {
    AsyncEvent inputReady = new AsyncEvent();
    AsyncEventHandler h = new AsyncEventHandler() {
        public void handleAsyncEvent() {
            System.out.print("The first Handler ran!\n");
        }
    };
    inputReady.addHandler(h);
    System.out.print("Test 1\n");
    inputReady.fire();
    Thread.yield();
    System.out.print("Fired the event\n");
}
```

AsyncEvent

```
SchedulingParameters low = new
    PriorityParameters(PriorityScheduler
        .getMinPriority(null));
inputReady.setHandler(new AsyncEventHandler(low,
    null, null, null, null) {
    public void handleAsyncEvent() {
        System.out.print("The low priority handler
            ran!\n");}});

SchedulingParameters high = new
    PriorityParameters(PriorityScheduler
        .getMaxPriority(null));
inputReady.addHandler(new AsyncEventHandler(high,
    null, null, null, null) {
    public void handleAsyncEvent() {
        System.out.print("The high priority handler
            ran!\n");}});
```

AsyncEvent

```
System.out.print("\nTest 2\n");  
    inputReady.fire();  
    System.out.print("After the fire\n");  
    Thread.sleep(100);  
    System.out.print("After the sleep\n");
```

AsyncEvent Output

Test 1

The first handler ran!

Fired the event

Test 2

The high priority handler ran!

After the fire

The low priority handler ran!

After the sleep

Timer

```
public class TimerExample {
    private static final
        SchedulingParameters highPriority = new
            PriorityParameters(PriorityScheduler.getMaxPriority(null));
    private static void TestTimer(String title, Timer t) {
        ReleaseParameters rp = t.createReleaseParameters();
        rp.setCost(new RelativeTime(10, 0));
        t.addHandler(new
            AsyncEventHandler(highPriority, rp, null, null, null) {
                public void handleAsyncEvent() {
                    System.out.print("  Timer went off at "
                        + (System.currentTimeMillis() - T0) + "\n");
                }
            });
        t.start();
    }
    //USE
    TestTimer("One Shot",
        new OneShotTimer(new RelativeTime(100, 0), null))
    TestTimer("Periodic",
        new PeriodicTimer(new RelativeTime(100, 0),
            new RelativeTime(100, 0), null));
}
```

AsynchronouslyInterruptedException

```
public void example() {
    MyInterrupt aie = new MyInterrupt();
    aie.doInterruptible(new Interruptible() {
        public void runNonInterruptible() {
            /* do something that can't be interrupted */
        }
        public void run(AsynchronouslyInterruptedException e)
            throws AsynchronouslyInterruptedException {
            /* This method can be interrupted at any point in time */
            runNonInterruptible();
            e.disable();
            e.enable();
        }
        public void interruptAction(
            AsynchronouslyInterruptedException e) {
            /* code which executes if run() method interrupted */
        }});
}
```