Real-Time Specification for Java™: The Revolution Continues

Bertrand Delsart, John Duimovich, Doug Locke

TS-5767

Java

Sun microsystems

Learn about the different flavors of Real-Time Java Specifications to find the one that best fits your tastes

GOAL

Who are these people?

- Bertrand : Sun Microsystems
  - Java RTS (Real-Time System) technical leader
  - Bertrand.Delsart@Sun.COM
- Doug: Locke Consulting LLC
  - Real-time systems consultant, JSR-302 Spec Lead
  - doug@douglocke.com
- John: IBM® Canada
  - Chief Technology Officer of the IBM Java Technology Centre
  - John_Duimovich@ca.ibm.com

Agenda

- Java Specification Request (JSR)-1: The Real-Time Specification for Java (RTSJ):
  - The Determinism of C/C++, a Taste of Java Code
- JSR-282: RTSJ version 1.1
  - Adding New Ingredients
- JSR-50: Distributed Real-Time Specification for Java
  - Adding Distribution Flavor
- JSR-302: Safety Critical Java
  - Simplifying the Recipe to Guarantee the Taste
- JSR-xxx
  - Revisiting the Taste

What is Real-Time ?

- Real-time isn't the same as real fast!
- Going faster helps...
- But what really matters is the Worst Case Execution Time

Real-Time is about deadlines !

- Military
  - e.g., accurate missile tracking; avoid getting blown up
- Telcos
  - e.g., predictable call connection; avoid irritating the user
- Banks
  - e.g., responsive trading; avoid not making money

Real-Time Use Case: Send Data at a Fixed Rate

- Mainstream Java Technology Solution:

```
while (true) do {
    compute_data();
    now = System.currentTimeMillis()
    Thread.sleep(next_period - now);
    send_data();
    next_period += period;
}
```

- Problem:
  - Not guaranteed to wake up quickly after the sleep call
    - Mainstream "`setPriority()`" is not sufficient
  - What if it is preempted just after the `currentTimeMillis()` call ?

JSR-1 Provides the Necessary APIs and Semantics

- Code executed by a RealTimeThread:

```
this.setPriority(my_RTPriority);
AbsoluteTime wakeup = ...;
RelativeTime period = ...;
while (true) do {
    compute_data();
    RealtimeThread.sleep(wakeup);
    send_data();
    wakeup.add(period,wakeup);
}
```

- Note on priority semantics:
  - It properly handles locks, boosting low priority threads if necessary
- Problem:
  - The system does not know the timing properties of this thread

JSR-1 Provides a Rich Real-Time Library

- Code executed by a RealTimeThread:

```
this.setPriority(my_RTPriority);
this.setReleaseParameters(myPeriodicParams);
while (true) do {
    compute_data();
    RealtimeThread.waitForNextPeriod();
    send_data();
}
```

- Advantage:
  - Richer semantic, with **Deadline Miss** monitoring and management and optionally Cost Enforcement and Feasibility Analysis
- Problem:
  - Is it sufficient to guarantee determinism ?

JSR-1 Provides Threads Optimized for Determinism

- Isolated from GC pauses
- Implicitly working around other non-deterministic optimizations (JIT compilation, recompilation, ...)

- 'Same' code... executed by a NoHeapRealtimeThread (NHRT)
- Result:
  - Programs running today as deterministic as C/C++
  - Java code flavor
    - Availability of a rich library for ease-of-use and portability
    - Lots of dynamic timing checks to easily detect and react to timing issues
- Problem:
  - What about memory allocation since 'isolated' from the GC ?

MemoryAreas for NoHeapRealtimeThreads

- ImmortalMemory for non recycled objects
- ScopedMemory for recycling
  - Memory areas not subject to the Garbage Collector
  - Per area counters to know how many threads have entered an area
  - Objects automatically deleted when the count of their area is 0
  - Tree of nested scope, dynamically modified by each enter() call
    - Reflecting which scopes can safely reference a given scope
    - With dynamic checks to ensure a “single-parent” rule
  - Dynamic read/write checks to guarantee safety
    - Read exception if the returned value is a Heap-allocated object
    - Write exception if a Scope-allocated object is stored in an object that may survive longer than the Scope

First NHRT Version: Try with the Same Code

- Code executed by a NoHeapRealtimeThread

```
while (true) do {
    scopedMemory1.enter(runnable);
    // scopedMemory1 recycled for the next loop
}
void run() {
    compute_data();
    RealtimeThread.waitForNextPeriod();
    send_data();
}
```

- Problem
  - Is compute_data() endorsing read/write constraints ?
  - More generally, what about legacy code, third party libraries... and even core libraries ?

Limiting Hard Real-Time to the Time Critical Part

- Part executed by the NoHeapRealtimeThread consumer

```
void run() { // in the consumer runnable
   RealtimeThread.waitForNextPeriod();
   send_data(scopes[consumer].getPortal());
}
```

- Part executed by the soft RealtimeThread producer

```
void run() { // in the producer runnable
   scopes[producer].setPortal(compute_data());
   RealtimeThread.waitForNextPeriod();
}
```

- Problem:
  - Soft real-time delays must not impact the hard real-time part
  - The user has to ensure the reference count drops to zero... but not too early
  - **`send_data()`** must enforce read/write constraints

Recovering from Production Delays

- softReleaseParameters.setDeadlineMissHandler(hardDMH);
- Bound No-Heap Deadline Miss Handler:

```
void handleAsyncEvent() {
   scopes[consumer].enter(dmh_runnable);
}
void run() {
   data = compute_simpler_data(); // deterministic
   scopes[consume].setPortal(data);
   softRTT.schedulePeriodic();
}
```

- Problem:
  - `compute_simpler_data()must` enforce read/write constraints

Using Two Scopes to Guarantee Recycling

- Part executed by the soft RealtimeThread producer

```
while (true) do {
   // produce in the current consumer scope
   producer = consumer;
   scopes[producer].enter(producer_runnable);
}
```

- Part executed by the NoHeapRealtimeThread consumer

```
while (true) do {
   // change to the other scope only if the
   // producer was not delayed inside it
   if (producer == consumer) {
      consumer = 1-consumer; // 0..1 toggle
   }
   scopes[consumer].enter(consumer_runnable);
}
```

JSR-1, RTSJ: The determinism of C/C++, a taste of Java Code

- Necessary Real-Time Extensions
- Rich Real-Time Library
- Interesting Time Related Checks
- A Mechanism to Avoid GC Pauses
- A Few Optional Features
  - Cost Enforcement, Feasibility Analysis, PCP Locks
- The Resulting Taste

  + Simple portable programs as deterministic as C/C++ ones
  + Powerful recovery mechanisms to improve robustness
  - Potential issues for more complex hard real-time tasks

Don't be put off !
With Real-Time Garbage Collectors (RTGC),
you can get the real taste of Java Code

Real-Time Garbage Collection: The Real Taste of Java Code

- Hard real-time consumer

```
while (true) {
   RealtimeThread.waitForNextPeriod();
   send_data();
}
```

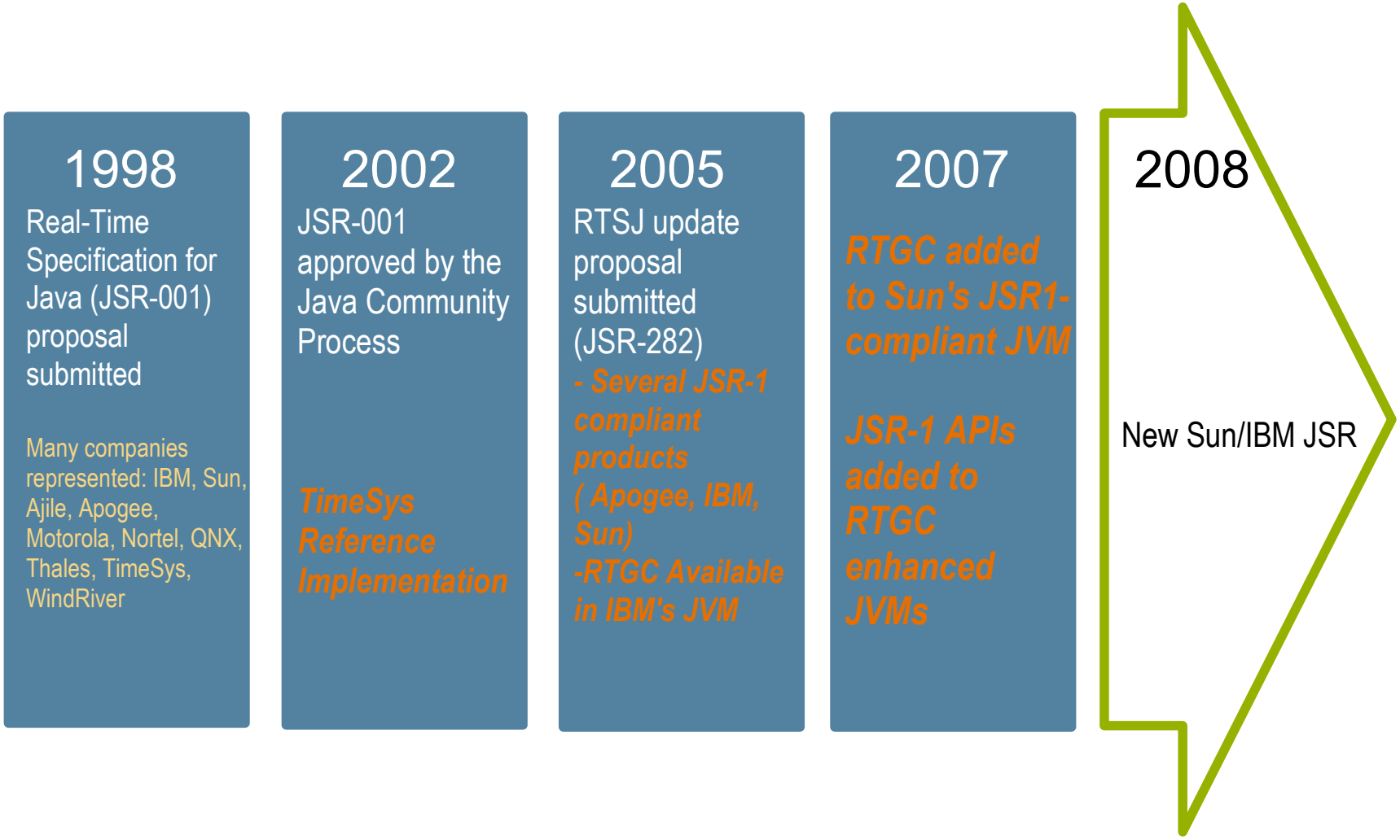
- Soft real-time producer

```
while (true) {
   RealtimeThread.waitForNextPeriod();
   compute_data();
}
```

- Hard real-time Deadline Miss Handler

```
void handleAsyncEvent() {
   compute_simpler_data();
   softRTT.schedulePeriodic();
}
```

JSR-1 Evolution



Agenda

- JSR-1: The Real-Time Specification for Java:
  - The Determinism of C/C++, a Taste of Java Code
- JSR-282: RTSJ version 1.1
  - Adding New Ingredients
- JSR-50: Distributed Real-Time Specification for Java
  - Adding Distribution Flavor
- JSR-302: Safety Critical Java
  - Simplifying the Recipe to Guarantee the Taste
- JSR-xxx
  - Revisiting the Taste

JSR 282: Further Enrich the RTSJ Library

- Title – "RTSJ version 1.1"
- Spec Lead: TimeSys Corporation, Peter Dibble leading
- Champion for each SI, "Specification Issue"

- Processor pinning:
  - Specify which CPUs a thread can use
  - No existing POSIX standard
  - Develop an API that is portable enough to work on various systems
- Consumed CPU time
  - Allow reasoning about CPU consumption instead of relying only on the optional cost enforcement
- Add data to fired events
  - Similar to cookies in POSIX signal handlers
- ...

JSR-282: Revisiting NHRT Memory Areas

- 1/3 of the Specification Issues concern memory
- 1/2 of them have already been closed
- Most of them are about ScopedMemory
  - Scoped Weak References, Pinned Scopes, Enhanced MemoryArea.enter method allowing to pass arguments, Removal of the bi-directional rule
- Significant progress on ImmortalMemory consumption
  - Will benefit class unloading and immortal memory 'leaks'
    - Option to prevent implicit ImmortalMemory allocations
    - Study of a new initialization strategy based on the area in which the ClassLoader was allocated being considered

JSR-282 progressing... slowly

- Initiated in 2005
- RI and TCK gradually enriched with new JSR-282 APIs and semantics
- A few Specification Issues are still being discussed due to their complexity
  - Feedback is requested to improve the proposal while it is still flexible
- RTGC technology changed the context; the most complex SIs are sometimes no longer the most important for customers because RTGCs allow them to be deterministic enough without using NHRT/ScopedMemory

Agenda

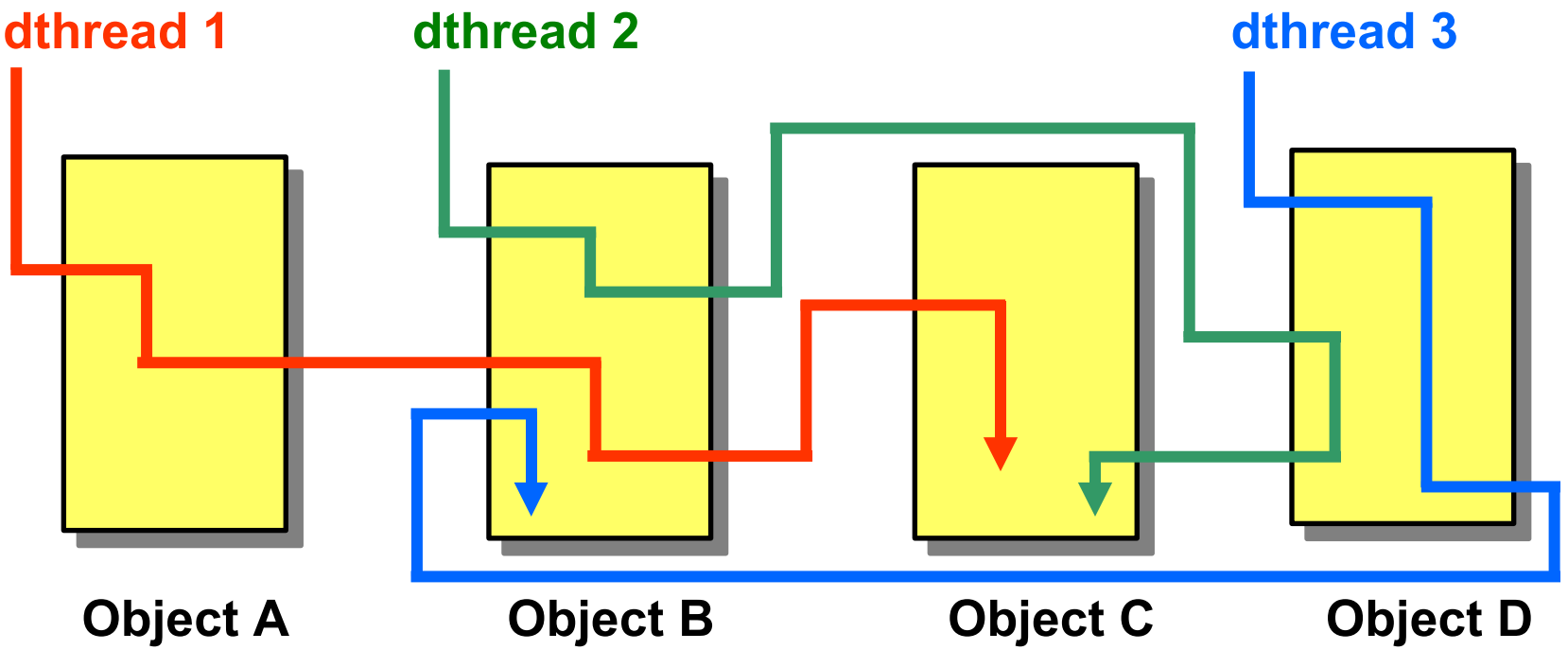
- JSR-1: The Real-Time Specification for Java:
  - The Determinism of C/C++, a Taste of Java Code
- JSR-282: RTSJ version 1.1
  - Adding New Ingredients
- JSR-50: Distributed Real-Time Specification for Java
  - Adding Distribution Flavor
- JSR-302: Safety Critical Java
  - Simplifying the Recipe to Guarantee the Taste
- JSR-xxx
  - Revisiting the Taste

JSR-50: Adding Distribution Flavor

- Title – "Distributed Real-Time Specification for Java"
- Spec Lead – Mitre Corp., Doug Jensen leading
- Bring distributed real-time support to Java technology
  - Provide support for end-to-end application timeliness and fault management properties
  - RTSJ applications intended to run unmodified on DRTSJ
  - Traditionally ensuring these end-to-end properties has been forced on the application designers
    - who must create ad-hoc (and error-prone) mechanisms to attain them
    - typically without proper experience or education
    - and at high recurring and non-recurring costs
  - Existing Java technology distribution models (e.g., Java Message Service, JXTA) do not provide appropriate end-to-end context…

JSR-50 Distributable Threads Components

- A *distributable threads* programming model
- A distributable *thread integrity* framework
- A scheduling framework



(Returns have not yet occurred)

JSR-50 Status

- Specification nearing completion
- Reference Implementation nearing completion
- Funding issues have caused a recent suspension of progress
  - Opportunities exist for interested parties to assist

Agenda

- JSR-1: The Real-Time Specification for Java:
  - The Determinism of C/C++, a Taste of Java Code
- JSR-282: RTSJ version 1.1
  - Adding New Ingredients
- JSR-50: Distributed Real-Time Specification for Java
  - Adding Distribution Flavor
- JSR-302: Safety Critical Java
  - Simplifying the Recipe to Guarantee the Taste
- JSR-xxx
  - Revisiting the Taste

JSR-302: Simplifying the Recipe to Guarantee the Taste

- Title – "Safety Critical Java Technology"
- Spec Lead: The Open Group, Doug Locke leading
- Goal – a specification for Safety Critical Java source capable of being certified under DO-178B Level A and other safety critical certification standards
  - The specification will be based on a subset of the Real-Time Specification for Java
  - Certification implies a small, reduced complexity infrastructure (i.e., Java Virtual Machine)
  - Emphasis is on defining a minimal set of capabilities required for safety critical applications

JSR-302 Overview

- Application Structure

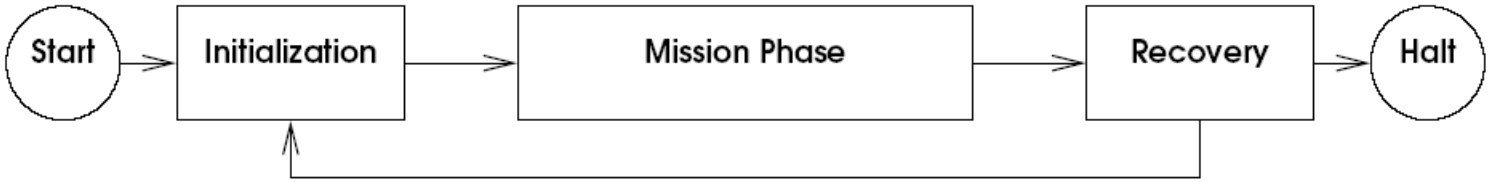


Figure 3.1: Safety Critical Execution Phases

  - Three Compliance Levels
  - Data structures created at initialization in Mission Memory
  - Scoped memory areas used for limited dynamic allocation
    - Restrictions on multiple memory area access
  - Application startup will not require heap memory
    - Uses a Safelet

JSR-302 Compliance Levels

- Three Compliance Levels
  - Level 0 provides a cyclic executive (single thread), no wait/notify
    - Synchronization ignored
    - No threads – only periodic Async Event Handlers
    - Local memory in local scoped memory – emptied each period
  - Level 1 provides a single mission with multiple schedulable objects, no wait/notify
    - Multiple concurrent schedulable objects
    - Dynamic scoped memory allowed, but not shared
  - Level 2 provides nested missions with (limited) nested scopes
    - May have NoHeapRealtimeThreads, wait/notify, nested shared scoped memory (must be statically checkable)
  - All Exceptions must be pre-allocated

JSR-302 Issues (1)

- Garbage Collector
  - None required
- Scoped memory
  - Nesting restricted to Level 2
  - Reference safety must be statically analyzable
- All Schedulable Objects will be non-heap
- Initialization initializes each class in user-defined order
- Java source memory model follows JSR-133
  - Circular reference initialization disallowed

JSR-302 Issues (2)

- Support for SMP's to follow RTSJ lead
  - May include Schedulable Object processor pinning
- No Finalizers
- No Reflection
- Requires Priority Ceiling for priority inversion management in Synchronized methods
  - No synchronized blocks other than "this"
- Priority Inheritance not required
- Class loader not required
- Raw memory included, but Physical Memory not required
- Required annotations to support static analyzability defined
- Specification will list required Java classes in conforming implementations

JSR-302 Status

- Specification draft writing assignments are almost completed
- Expect initial specification soon
- Reference Implementation being implemented as open source RTSJ-compliant Java code executable on any RTSJ-compliant JVM™
- Technology Compatibility Kit still to be worked
- Strong Expert Group
- Stay tuned!

Agenda

- JSR-1: The Real-Time Specification for Java:
  - The Determinism of C/C++, a Taste of Java Code
- JSR-282: RTSJ version 1.1
  - Adding New Ingredients
- JSR-50: Distributed Real-Time Specification for Java
  - Adding Distribution Flavor
- JSR-302: Safety Critical Java
  - Simplifying the Recipe to Guarantee the Taste
- JSR-xxx
  - Revisiting the Taste

JSR-xxx, Why: Avoid Real-Time Java Technology Fragmentation

- JVM implementation issues with ScopedMemory
  - ScopedMemory is hard to implement efficiently
  - ScopedMemory enhancements are very intrusive
  - Each implementor must decide which classes must be modified to be Scope-safe
  - A few issues are still being worked on in JSR-282
- Alternative approaches sufficient for many customers
  - RTGC technologies, included in most of the RT JVMs
  - Other proprietary extensions that permit smaller latencies in a subset of the application (xRTs for instance)
- Consequence
  - A few vendors do not try to be fully JSR-1 compliant
  - Users have more portability issues

JSR-xxx, How: Subset that's Useful and Easy

- Proposed during “Future Directions for the RTSJ” BOF, at JTRES 2007 (Workshop on Java Technologies for Real-time and Embedded Systems)
  - http://www.vmars.tuwien.ac.at/jtres2007/slides/BoF.pdf
  - No objections raised from the RT experts at the workshop
  - Proposal removed memory management and ATC
- Focus on memory management to proceed faster
- Mostly paperwork to define:
  - A new configuration, subset of JSR-1:
    RTSJ Configuration for Alternative Memory Allocators
  - A profile defining the current JSR-1 memory management
    NoHeapRealtimeThread profile for RTSJ
- Leveraging JSR-1 RI and TCK

JSR-xxx, Who:

- Possibly co-led by IBM and Sun
- TimeSys delivering the RI and the TCK
- Expert Group being formed to prepare submission into JCP℠ service
- You can still be part of it !

Summary

- JSR-1 products already exist and are successfully used
- JSR-282 is continuing to improve RTSJ
- JSR-50 and JSR-302 will extend the real-time Java technology market
- JSR-xxx should increase the compliant real-time JVMs offering

- You can still contribute to these specification efforts
  - Go to http://jcp.org/ for the existing JSRs
  - Contact Bertrand.Delsart@Sun.COM for the new one

THANK YOU

Bertrand Delsart, John Duimovich, Doug Locke